Multilevel Parallelism: Case Studies and Lessons Learned

Daniel Duffy[1,2] and Mark Fahey[3]

[1]Computational Grid Group
[2]Computer Science and Engineering Group
Computer Sciences Corporation
ERDC MSRC
Vicksburg, MS

[3]Center for Computational Sciences
University of Tennessee
Oak Ridge National Laboratory
Oak Ridge, TN

SC2001: November 10-16, 2001

Outline

- ERDC MSRC
- ORNL
- Introduction
  - Parallel Strategies
  - Processes versus Threads
  - Hybrid Model
- OpenMP and Pthreads Overview
- Example: Calculating Pi

Outline (continued)

- Tools
  - TotalView
  - Vampir
  - KAP Pro Tools
- Implementing Mixed-Mode Codes
- Case Studies
  - STWAVE
  - CGWAVE
  - SARA-3D
  - FEMWATER123
- Lessons Learned
- Future
- 1-D Laplace Example

ERDC MSRC http://www.erdc.hpc.mil

- U.S. Army Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS
  - Hardware
    - 2 Compaq machines (soon): 256 ES40 and a 512 ES45
    - Origin 3800: 512-processor single-system image
    - IBM Power3 SMP: 16 nodes (8-way); leaving soon
    - Cray T3E: recently upgraded
      - 512 600 MHz processors with 256 Mbytes of memory
      - 256 675 MHz processors with 512 Mbytes of memory

  - Computational Science and Engineering Group
    - Computer Scientists, mathematicians, earth scientist, physicist
    - Benchmarking
    - Porting/Parallelizing/Optimizing code
    - Algorithm development and research
    - Tools development
    - Education
  - Computational Grid Group
    - Developing a Computational Grid for the DoD based on Globus

ORNL
http://www.ccs.ornl.gov

- DoE High Performance Computing Research Center at Oak Ridge National Laboratory, Oak Ridge, TN
  - Hardware
    - IBM Power4: 24 nodes each with 32 1GHz processors being delivered
    - IBM Power3: 176 4-way and 8 2-way 375 GHz
    - Compaq SC40: 64 node 4-way 667 MHz
    - Compaq SC40: 16 node 4-way 667 MHz
  - Computer Science and Mathematics Division
    - High performance computing
    - Applied mathematics
    - Intelligent systems
    - Advanced computing systems
    - New ways to solve problems beyond the reach of most computers
    - Software tools

What is Multilevel Parallelism?

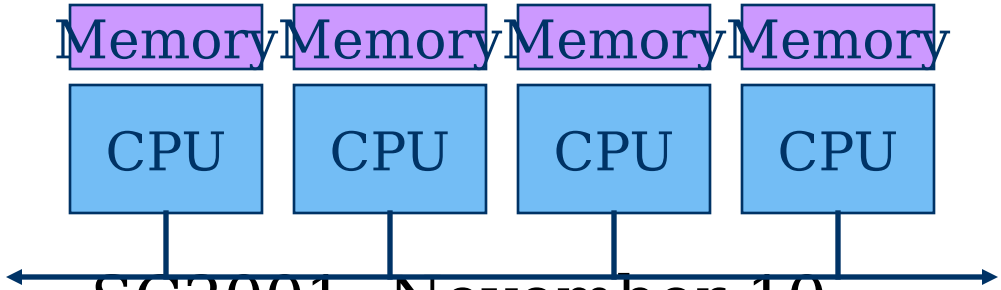
- Often referred to as dual-level parallelism, mixed-mode, or hybrid programming model
- Contains some mixture of at least two of the following:
  - MPI
  - OpenMP
  - Pthreads
  - Shmem
  - Fork
  - Other
- In general, multilevel parallelism is the mixture of at least two methods of parallel paradigms in the attempt to create a program that scales better and runs faster than can be obtained with only a single method.

Why Multilevel Parallelism?

- Memory limitations
  - Some shared-memory nodes may not have enough memory for x copies of the executable.
- Intensive operations
  - Each MPI process contains a large number of computations.
  - Speed these areas up with threads.
- Ease of use
  - Using threads to perform loop-level parallelism within MPI processes is relatively easy.
  - Usually, the number of source code lines that needs to be added is generally small.
- Portability
  - With OpenMP, directives can be ignored at compile time.
  - Follow the standard, either OpenMP or Pthreads.
- Disk limitations
  - If large amounts of I/O are needed for each MPI process, threads can allow the number of I/O requests to be kept to a minimum.

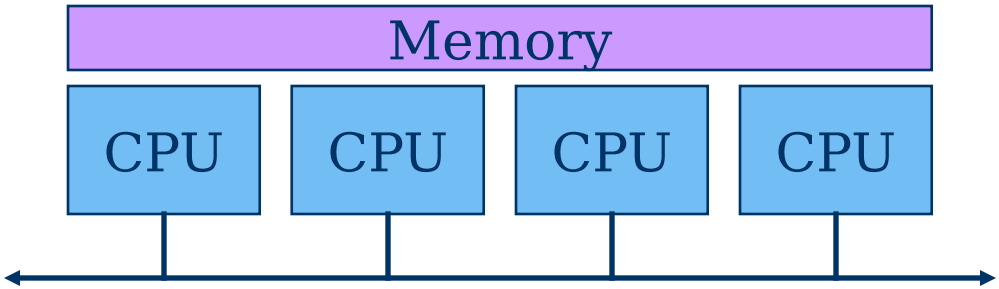
Architecture: Distributed Memory

- Each node contains a single processor with its own memory.
- Nodes are tightly coupled via switches or routers.
- Classic Examples
  - T3E: Nodes positioned within a torus architecture
  - IBM SP: Nodes connected through switches



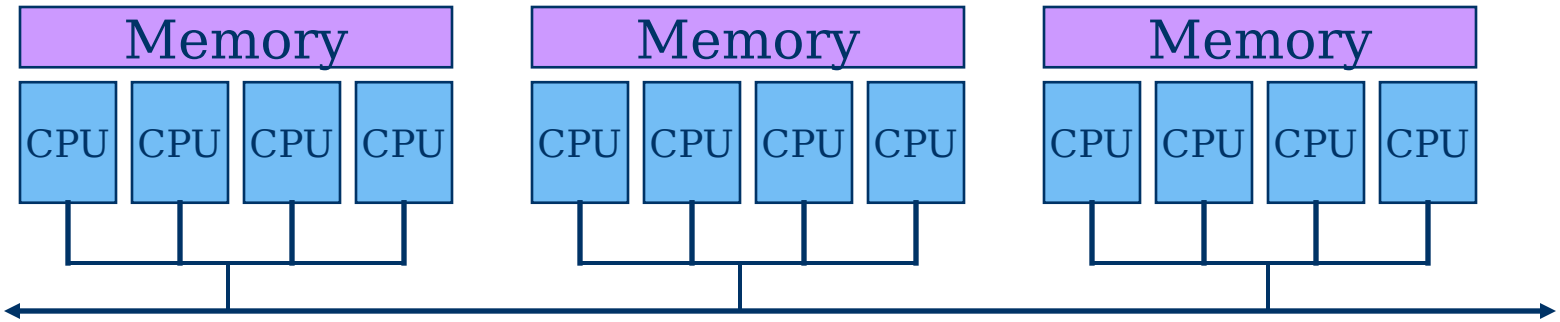
Architecture: Shared Memory Processors (SMP)

- N processors all share a single address space: Single System Image (SSI)
- Path to the memory is either
  - Uniform Memory Access (UMA): The same for each processor
  - Non-Uniform Memory Access (NUMA): Different depending on the physical position of the processor
- Examples
  - SGI Origin Series
  - Sun E10K



Architecture: Distributed-Shared Memory Processors (DSMP)

- Each node contains n processors that share the memory on that node only.
- Many nodes are loosely connected.
- Examples
  - IBM Power3, Compaq SC40 and SC45
  - IBM Power4, Compaq GS320



Message Passing Interface (MPI)

- Standard for message passing
  - MPI Standard 1.1
  - MPI Standard 2.0
- MPI is a library specification, not a language
- Model: A collection of processes communicating via messages
- Can be used on almost every architecture
  - Vendor-supplied libraries
  - MPICH
  - Other implementations
- http://www.mcs.anl.gov/mpi/

OpenMP

- Application Program Interface (API) that supports multi-platform shared-memory parallel programming in C/C++ and Fortran
- Available on most architectures, including Unix platforms and Windows NT platforms
- Standard for shared-memory programming
  - Fortran 2.0
  - C/C++ 1.0
- Comprised of
  - Compiler directives
  - Environment variables
  - Library routines
- Excellent Web site at http://www.openmp.org/

Pthreads

- Posix C API thread library: IEEE POSIX 1003.1c standard (1995)
- Created to provide a standard for vendor-specific threading capabilities
- Fortran interface
  - ERDC MSRC
    - http://www.erdc.hpc.mil/hard_soft/pthreads.htm
  - IBM xlf7.1
- Nice tutorial http://www.llnl.gov /computing/tutorials/workshops/workshop/pthreads/MAIN.html

Other Common Strategies

- Shmem
  - Originally the Cray message-passing library
  - Small number of routines
  - Single-sided
  - Very fast
  - Not always portable
- Fork
- Automatic Parallelization: -apo
  - Most vendors supply an option to the compiler
  - Use with caution
  - Not very sophisticated

Parallel Strategies: Task Level (1)

- Task level parallelism at the input data level, sometimes called embarrassingly parallel
  - Relatively easy to do
  - Typically, minimal message passing
  - Executable may not fit on a node
    - If a.out uses x amount of memory, mpirun –np y will use x*y amount of total memory.
    - This may be infeasible on some shared-memory nodes. For example, the IBM SMP at ERDC has only 4 Gbytes of memory per 8-way node.

ERDC MSRC Major Shared Resource Center

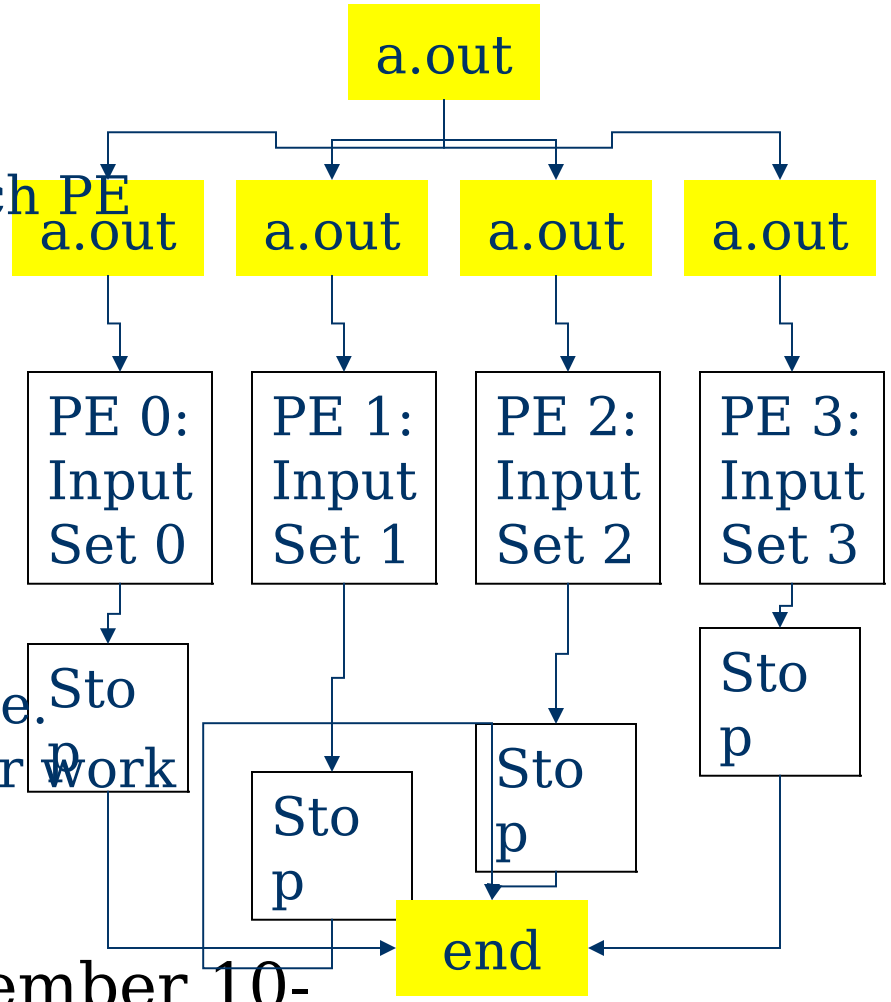
Parallel Strategies: Task Level (2)

mpirun –np 4 a.out

Size of executable for each PE remains the same.

Each processor works on a unique input set. Little or no communication.

Load Balancing is an issue. Some PEs may finish their work before others.



Parallel Strategies: Domain Level (1)

- Parallelism by domain decomposition or substructuring
  - Relatively difficult to do
  - Usually message-passing intensive
  - The size of the executable usually shrinks as the number of processors used grows.
    - If a.out uses x amount of memory, mpirun –np y will typically use x amount of total memory.
    - Per process, a.out will use roughly x/y amount of memory.

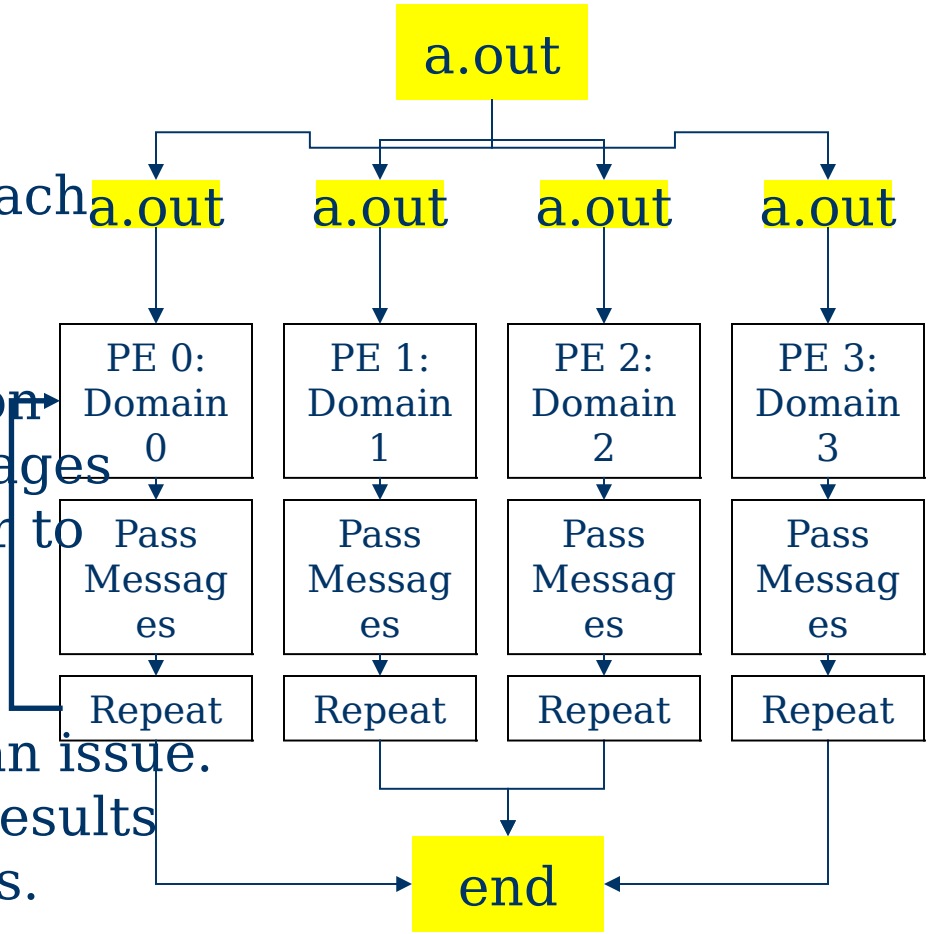
Parallel Strategies: Domain Level (2)

mpirun –np 4 a.out

Size of executable for each PE is smaller.

Each processor works on a unique domain. Messages must be passed in order to communicate results.

Load Balancing is still an issue. PEs are dependent on results obtained from other PEs.



Processes versus Threads
MPI versus OpenMP/Pthreads

- Processes
  - Contain information about program resources and execution state, e.g.
    - IDs
    - Stack
    - Registers
  - Have their own address space
  - Can have multiple threads
- MPI
  - Many processes
  - Shared-nothing architecture
  - Explicit messaging
  - Implicit synchronization
  - All or nothing parallelism

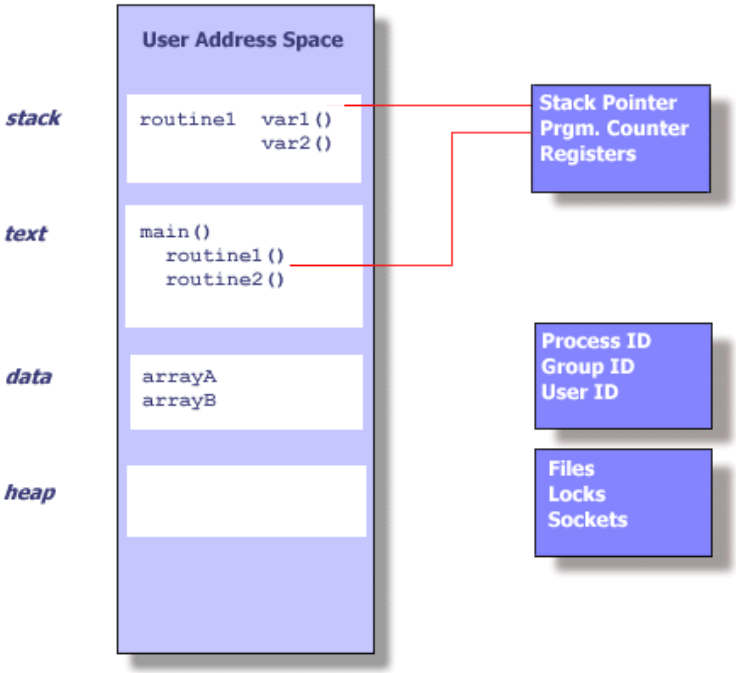
- Threads
  - Execute within a process
  - Same address space
  - Share Process's stack space
  - Thread-specific data

- OpenMP/Pthreads
  - 1 process, many threads
  - Shared-everything architecture
  - Implicit messaging
  - Explicit synchronization
  - Incremental parallelism

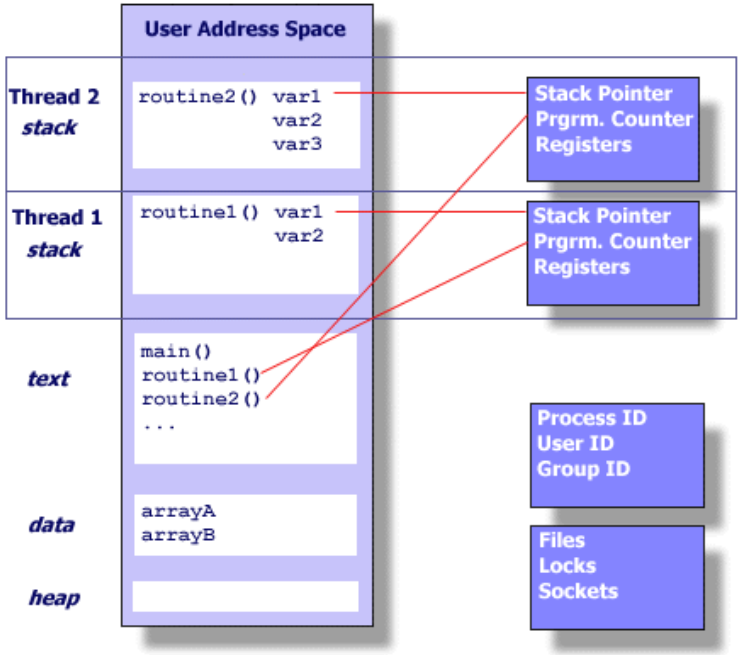
ERDC MSRC Major Shared Resource Center

Processes versus Threads (Continued)

Processes



Threads



Taken from
http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.ht

MPI versus Threads

Functionality	MPI	Threads
Ease of Use:	Harder	OpenMP: Easy Pthreads: Harder
Portability:	Standard[1]	OpenMP: Standard[2] Pthreads: Posix
Overhead:	Large	Small
Capability/Functionality:	Very High	High

[1]Most vendors do not support all of the MPI-2 standard, though many of the MPI-IO functionality is supported.
[2]Though a standard, not all functionality may be supported (it usually is, but be careful).

ERDC MSRC Major Shared Resource Center

OpenMP versus Others

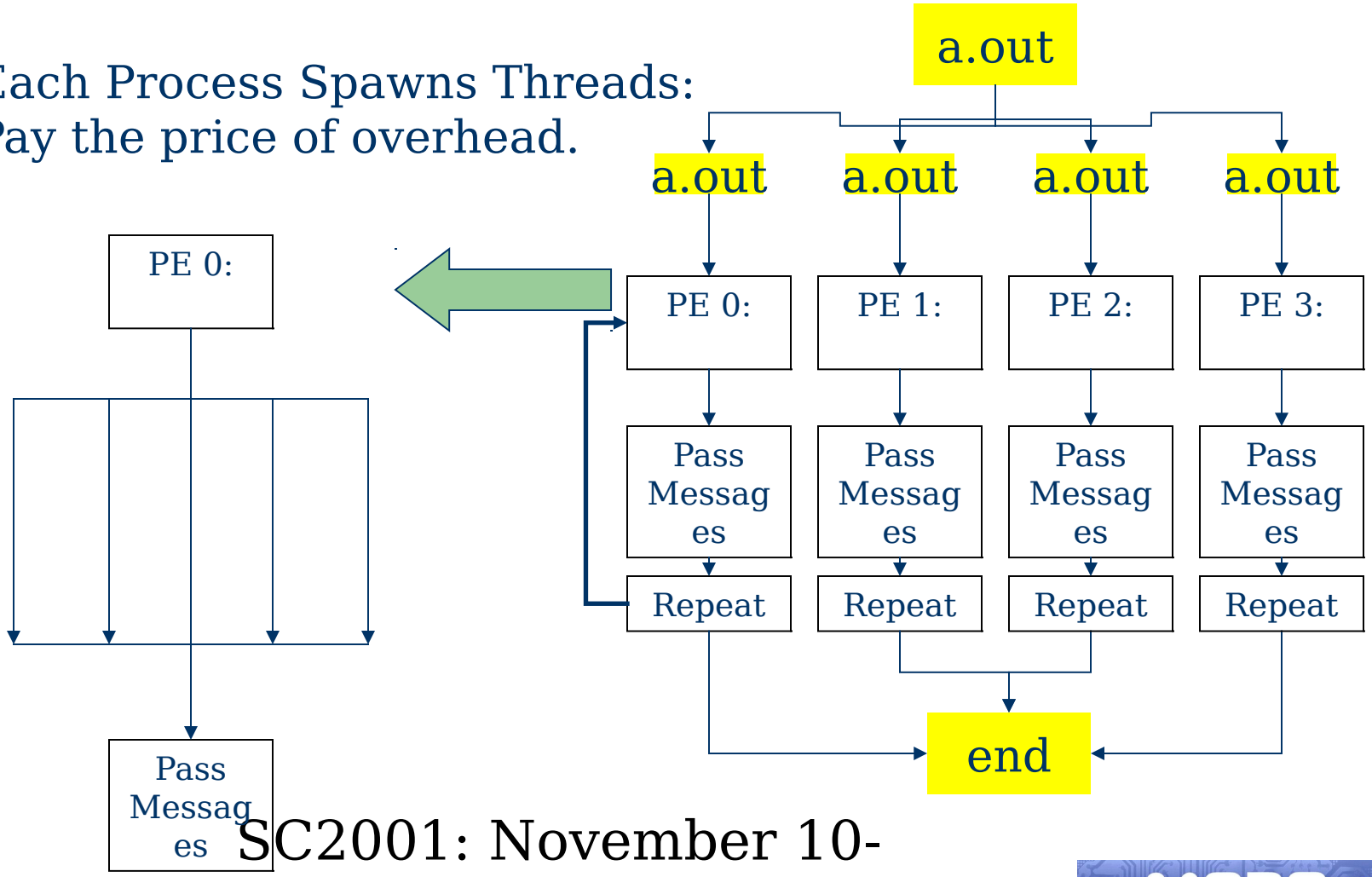
Thread Type	Architectures	Ease of Use	Overhead	Portabi
OpenMP	SMPs Distributed[1]	Very Easy	Small	High
Pthreads	SMPs	Hard	Small	High
Fork	SMPs	Not Easy	Larger	Low[2]
Vendor	Specific	Depends	Small None	

[1]Coming soon.
[2]Stick to the Posix standard calls and the portability will be high. However, fork calls may not be the most portable solution for all architectures.

SC2001: November 10-16, 2001

Parallel Strategies: Multilevel

Each Process Spawns Threads:
Pay the price of overhead.



Hybrid Programming Model

- Message passing across nodes
- Shared memory within a node
  - Explicit via POSIX threads or Fork
  - Implicit via SMP compiler (e.g., OpenMP)
- Usually only one thread calls MPI
  - Fork after and join before MPI calls
  - In general, use threads within MPI rather than the other way around
- Can provide better scalability than either MPI or threads alone
- Note: Most of this tutorial will focus on MPI+OpenMP.

ERDC MSRC Major Shared Resource Center

A Few Things to Remember

- Easier to use while still maintaining the portability of the code
- OpenMP currently only works on shared-memory machines, while MPI works on almost all platforms.
- OpenMP works best with loops that perform large amounts of computation.
- MPI works best when parallelizing code on a more coarse grain.
- Threads are more “light weight” than processes and have a smaller overhead.
- OpenMP and MPI can be combined to create a multilevel approach.
  - Use MPI to parallelize on a large scale and OpenMP within MPI to break up the work.

OpenMP Overview

SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

OpenMP Parallel Model

- Aimed at data parallelism
- Loop level parallelism is usually limited in its scalability
- OpenMP directives are processed by the compiler to generate code that can be executed in parallel
- OpenMP programs follow a fork-and-join model
  - A single thread exists at the beginning (master).
  - Upon entering a parallel region, a team of threads is spawned.
    - Execution is replicated by all threads until a worksharing construct is encountered.
    - Worksharing construct indicates work is to be distributed among threads.
  - At the end of the parallel region, the execution returns to the master thread.

ERDC MSRC Major Shared Resource Center

Parallel/Worksharing Constructs

- *parallel*
  - Defines a region of the program to be executed by multiple threads in parallel; fundamental construct
- Worksharing constructs are used within a parallel region in order to break up the work between threads
  - *do* (Fortran) or *for* (C)
    - Identifies an iterative work-sharing construct where the iterations are distributed across threads
  - *sections*
    - Specifies that a set of constructs (*section*) is to be divided among a team of threads
  - *single*
    - The associated code block is to be executed by only one thread (not necessarily the master)

Worksharing Construct Notes

- A worksharing directive not lexically enclosed in a parallel region is considered *orphaned* and is distributed among team members of the dynamically enclosing parallel region.
  - Can simplify task of implementing coarse-grained parallelism
- Threads wait until all team members have arrived at the end of a parallel region.
  - The team is disbanded and master thread continues.
  - The *nowait* option can be used to specify that threads do not wait at the end of a worksharing construct.

ERDC MSRC Major Shared Resource Center

Scheduling Options

- Can specify the iteration scheduling for a parallel do construct with
  - the *schedule* clause
  - OMP_SCHEDULE environment variable
  - Options
    - Static
    - Dynamic
    - Guided
    - Runtime
      - Specify chunksize with first three

ERDC MSRC Major Shared Resource Center

Synchronization Constructs

- *critical*
  - Restricts execution to a single thread at a time
- *barrier*
  - Synchronizes all threads in a team
- *flush*
  - Specifies a “cross-thread” sequence point at which all threads in a team must have a consistent view of certain objects
- *atomic*
  - Ensures that a memory location is updated atomically, allows for better optimization than critical

Synchronization Constructs (cont.)

- *master*
  - Specifies that execution may only be performed by the master thread of the team
- *ordered*
  - Execution is performed in the order in which iterations would be executed in a sequential loop.

ERDC MSRC Major Shared Resource Center

Data Sharing

- Data sharing must be specified at the start of a parallel region with the *shared* and *private* clauses.
  - Shared variables are shared among all threads of a team.
  - Private variables are private to a thread.
  - The *firstprivate* clause initializes private copies of variables from the original object.
  - The *lastprivate* clause provides that the thread that executed the sequentially last iteration to update the object it had before the construct.
- The *threadprivate* directive privatizes common blocks
  - The *copyin* clause applies only to common blocks declared as threadprivate.
    - Specifies that the data in the master thread be copied to the thread private copies of the common block

ERDC MSRC Major Shared Resource Center

Reduction Clause

- Perform a reduction on a set of variables within a list
- Similar to the MPI_Reduce function call
- Operators
  - +, *, -
  - MAX, MIN, IAND, IOR, IXOR

ERDC MSRC Major Shared Resource Center

Compiling an OpenMP Code

- OpenMP uses compiler directives to parallelize your code. Not be used unless specified at compile time.
  - F77: c$OMP
  - F90: !$OMP
  - C: #pragma omp
- How do you compile to use the OpenMP directives?
  - SGI
    - cc *program.c* –mp
    - f90 *program.f* –mp
    - f77 *program.f* –mp –mpio

Major Shared Resource Center ERDC MSRC

Running an OpenMP Code

- How do you specify the number of OpenMP threads you want to use?
  - Environment Variables
    - csh: setenv OMP_NUM_THREADS 4
    - ksh: export OMP_NUM_THREADS=4
  - Built-in OpenMP functions allow you to specify at runtime the number of threads.
- How do you run an OpenMP code?
  - SGI: ./a.out (nothing special)

OpenMP Tips

- Focus on loops where the codes spend the most time, remove cross-iteration dependencies if possible
- Merge adjacent parallel dos
- Scheduling
  - Set scheduling type and/or chunk size for best performance
- Synchronization tuning
  - Use environment variables to tell threads how to wait for synchronization events.
- Data placement
  - First touch policy
  - SGI has an alternative round robin data allocation.
- Locking threads to CPUs
  - SGI has mustrun env var
- Data dependencies and cache use very important

ERDC MSRC Major Shared Resource Center

Pthreads Overview

SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

Pthreads Parallel Model

- POSIX standard to control the spawning, execution, and termination of tasks within a process
- Used primarily for task level parallelism
- More complex than OpenMP (and MPI)
  - Programming model
- Lower level than OpenMP (explain)
  - More control over scheduling and synchronization
  - Can control the grain size of each thread

Pthreads Model

- Threads
  - Have local/private memory
  - Share the memory space of the global process
  - Each has an instruction stream
- On SMPs, the system can run threads in parallel
- Threads are called “lightweight” processes
  - Only small portions of memory are needed when switching context between threads

ERDC MSRC Major Shared Resource Center

Advantages of Pthreads

- In comparison to multiple processes
  - Resource use
    - Have access to all resources of the process
  - Shared memory for communication
    - Not swapped out for thread context switch
    - Use shared variables for communication
      - Cost of two memory accesses
  - Low overhead
    - Much faster to create, synchronize, or context switch
  - Nested parallelism
    - Any thread may create new threads
    - All threads are peers

ERDC MSRC Major Shared Resource Center

Communication Models

Threads

- Data are shared in global memory space
  - Must be careful of read/write, write/write conflicts
- Explicit synchronization needed to avoid race conditions
  - i.e., concurrent access of same variable by multiple threads

Message-passing

- Parallel tasks share data through explicit messages
- Messages become synchronization points

Concurrent Programming

- Must consider all possible execution paths
  - May be running on same processor
  - May be running on different processors
  - Processors may have different speeds
- Different output from one run to the next
- Must use some form of synchronization
- Programming requires some skill
- Traditional applications
  - ATM network
  - Database search

ERDC MSRC Major Shared Resource Center

Pthread Basic Function

- Pthread_create
  - Create a thread
- Pthread_join
  - Wait for thread to finish (barrier)
- Pthread_exit
  - Halt execution of calling thread

ERDC MSRC Major Shared Resource Center

Synchronization with Pthreads

- Mutex – mutual exclusion
  - Enforces single thread access to a critical section
  - Enables correct programming execution
  - Mechanism is a lock (mutex)
    - Atomic operations
    - Only single thread can “hold” mutex at any time
- Pthread_mutex_init
- Pthread_mutex_lock
- Ptherad_mutex_unlock
- Pthread_mutex_destroy

ERDC MSRC Major Shared Resource Center

E-mail Addresses

- Daniel Duffy:
  - Daniel.Q.Duffy@erdc.usace.army.mil
- Mark Fahey:
  - MarkRFahey@yahoo.com

Pi Example

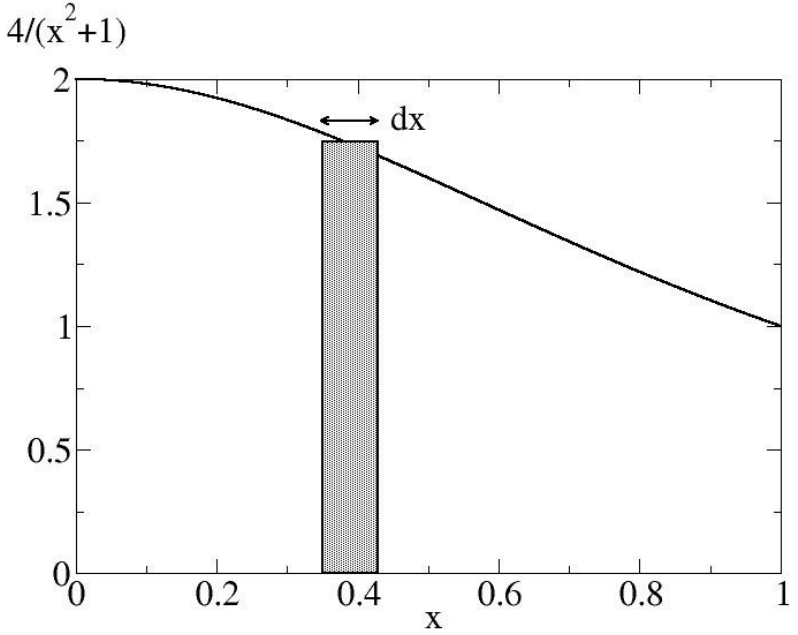
SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

Equation for Pi

There are many ways in which to compute the value of pi. One easy way to program is by the following integral:

$$\int_0^1 \frac{dx}{x^2+1} = \frac{\pi}{4}$$



Source Code: Serial Version

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main (int argc, char *argv[]) {
  const int    N  = 100000000;
  const double h  = 1.0/N;
  const double PI =
3.14159265358979323846264 3;
  int i;
  double x,sum,pi,error,time;

  time = START_TEST();

  sum = 0.0;
  for (i=0; i<=N; i++) {
     x = h * (double) i;
     sum+=4.0/(1.0+x*x);
   }
  pi = h*sum;

  time = STOP_TEST();

  error = pi-PI;
  error = error<0 ? -error:error;
  printf("pi = %18.16f +- %18.16f
\n",pi,error);
  return 0;
}
```

User Defined Timers!

Sum the Integrand.

Stop timer and print answer.

ERDC MSRC Major Shared Resource Center

OpenMP Pi

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main (int argc, char *argv[]) {
  const int    N  = 100000000;
  const double h  = 1.0/N;
  const double PI =
      3.14159265358979323846264 3;
  int i;
  double x,sum,pi,error,time;

  time = START_TEST();

  sum = 0.0;
#pragma omp parallel for
        shared(N,h) private(i,x)
      reduction(+:sum)
  for (i=0; i<=N; i++) {
     x = h * (double) i;
     sum+=4.0/(1.0+x*x);
    }
  pi = h*sum;

  time = STOP_TEST();

  error = pi-PI;
  error = error<0 ? -error:error;
  printf("pi = %18.16f +- %18.16f \n",pi,error);
  return 0;
}
```

OpenMP Directive!

Note: The shorthand method fo
the omp parallel construct was

Alternative:
#pragma omp parallel
#pragma omp for

ERDC MSRC Major Shared Resource Center

Pthreads Pi

```
#include <pthread.h>
#include <stdio.h>
#include <math.h>
#include "timer.h"

pthread_mutex_t reduction_mutex;
pthread_t *tid;

const int        N  = 100000000;
const double  PI =
      3.14159265358979323846264643;
int num_threads = 6;
double           pi = 0.0;

void *Piworker(void *argc) {
  int i, myid;
  double sum, mypi, x;
  double h  = 1.0/N;
  /* set id to start at 0 */
  myid = pthread_self()-tid[0];
  /* integrate function */
  for (i=myid+1; i<=N; i+=num_threads) {
      x = h*((double) i);
      sum = sum + (4.0 / (1.0 + x*x)); }
  mypi = h*sum;
  /* reduce value */
  pthread_mutex_lock(&reduction_mutex);
  pi += mypi;
  pthread_mutex_unlock(&reduction_mutex);
  return(0);
}
```

```
int main (int argc, char *argv[])
{
  int i;
  double error,time;

  time = START_TEST();

  pi = 0.0;
  tid = (pthread_t *) calloc(num_threads,
       sizeof(pthread_t));
  pthread_mutex_init(&reduction_mutex,
       NULL);

  for (i=0; i<num_threads; i++)
     pthread_create(&tid[i], NULL, Piworker,
       NULL);

  for (i=0; i<num_threads; i++)
    pthread_join(tid[i], NULL);

  time = STOP_TEST();

  error = pi-PI;
  error = error<0 ? -error:error;
  printf("pi   = %18.16f +- %18.16f \n",pi,error);
  return 0;
}
```

SC2001: November 10-16, 2001

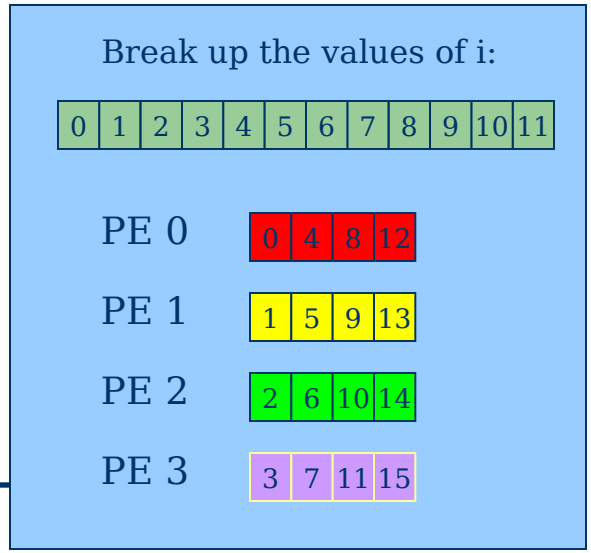
How do we use MPI?

- Simply break up the loop and allow the various MPI processes to execute different pieces of the loop.
  - Need to know the rank.
  - Need to know how to break up the loop.
  - Gather the individual pieces of the sum to a single process and verify the correct answer.
  - Use timers to see the speed up for different numbers of processes.
  - Scale the size of the problem, N.

ERDC MSRC Major Shared Resource Center

Source Code: MPI Version

```
... Declare variables and include mpi.h ...

  /* Initialize MPI.*/
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  time = START_TEST();
  sum = 0.0;
  for (i=myrank; i<=N; i=i+nproc) {
     x = h * (double) i;
     sum+=4.0/(1.0+x*x); }
  mypi = h*sum;
  pi   = 0.0;

  MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,
           MPI_SUM,0,MPI_COMM_WORLD);
  time = STOP_TEST();

... Compute and print the error from process 0 ...
... Finalize and return ...
```



Sum Local Values of Pi.

Source Code: Mixed-Mode Version

```
... Declare variables and include mpi.h
...
... Initialize MPI ...

  time = START_TEST();
  sum = 0.0;
#pragma omp parallel for
        shared(N,h,myrank,nproc)
        private(i,x) reduction(+:sum)
  for (i=myrank+1; i<=N; i=i+nproc) {
      x = h * (double) i;
      sum+=4.0/(1.0+x*x);
    }
  mypi = h*sum;
  pi   = 0.0;

  MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,
MPI_SUM,0,MPI_COMM_WORLD);
  time = STOP_TEST();

... Compute and print the error from
process 0 ...
... Finalize and return ...
```

Include the OpenM directives to sum t local values of Pi using threads.

Sum Local Values of Pi.

Resulting Timings

- Timings taken on an Origin 2800, 200 MHz
- N = 100,000,000
- Serial Code takes 25.13 seconds
- All times in seconds
- Note: Did not run the multilevel code.

NPs	MPI	OpenMP	Pthreads
1	25.14	25.17	25.14
2	12.65	12.60	12.58
3	8.40	8.46	8.38
4	6.28	6.34	6.29
5	5.04	5.06	5.03
6	4.20	4.23	4.19

ERDC MSRC Major Shared Resource Center

Task Scheduling: Static

Static assignment of tasks: Processors have a deterministic set

Worker 1	Task 1	Task 5	Task 9
Worker 2	Task 2	Task 6	Task 10
Worker 3	Task 3	Task 7	Task 11
Worker 4	Task 4	Task 8	Task 12

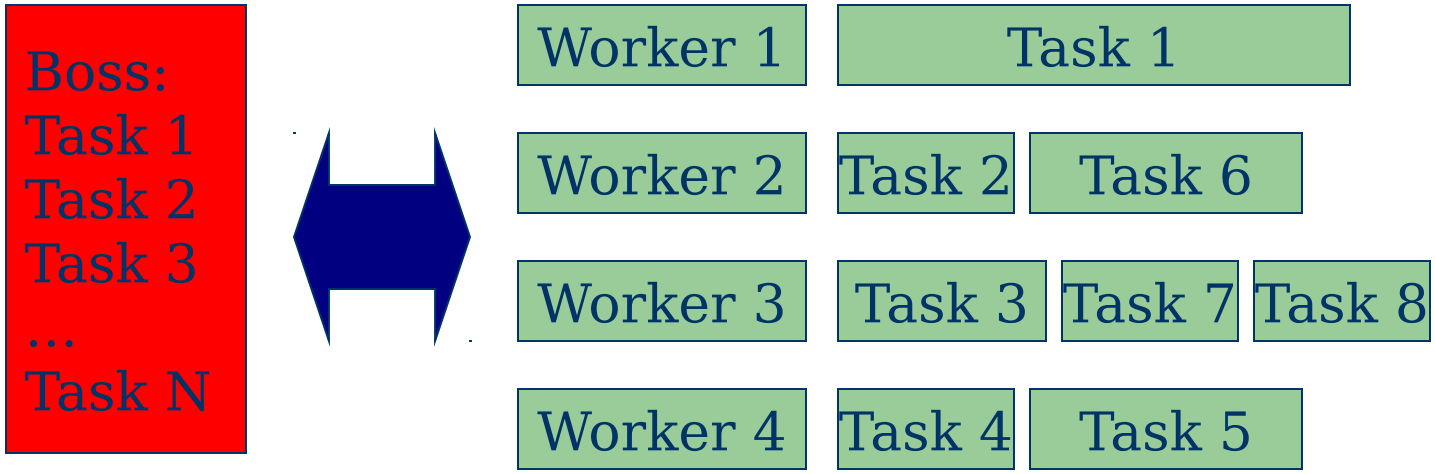
Disadvantages: Unequal sized tasks can lead to a problem with
Advantages: Equal sized tasks make it very easy to run with go
balancing. No boss, so all processes do work.

ERDC MSRC Major Shared Resource Center

Task Scheduling: Dynamic

Dynamic assignment of tasks: A processor gets a new task whe
finishes with its current task.

Boss-Worker is the classic example.



Advantage: Better load balancing with unequal tasks.
Disadvantage: The boss does no work, so one processor is lost.
always.)

Load Balancing

- Good load balancing minimizes the amount of time each processor is waiting for another processor, either for data or some other type of synchronization.
- Extremely important, and not easy to do.

ERDC MSRC Major Shared Resource Center

Tools

SC2001: November 10-16, 2001

TotalView

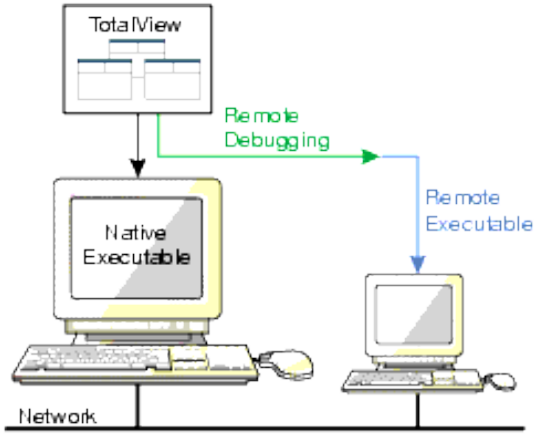
- http://www.etnus.com
- Version: 5.0
- Platforms
  - Compaq Alpha, Sun Sparc and SunOS 5, IBM RS/6000 and SP Power, SGI, HP, x86
- Support for
  - Threads/OpenMP
  - MPI/PVM
  - C/C++ (including nested templates), Fortran 77, Fortran 90/95 (including modules), HPF
- (Note there is also a Cray TotalView.)

TotalView Features

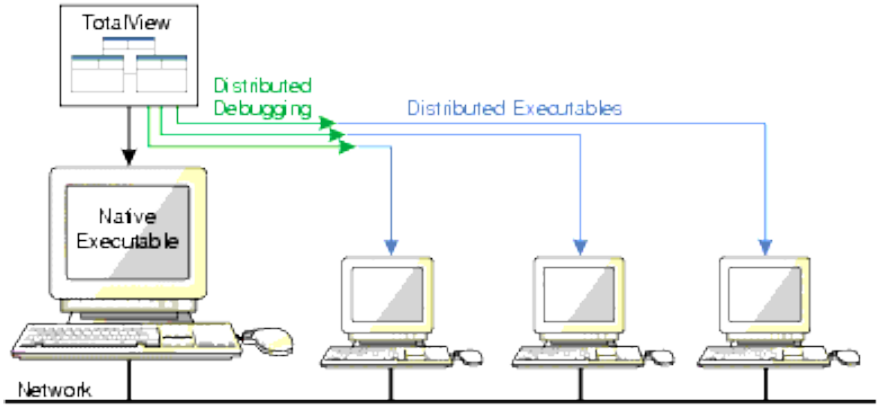
- Graphical User Interface and a Command Line Interface
- Handles multiprocess multithreaded programs
- TotalView runs on all major UNIX platforms so you can use the same debugger regardless of where you are debugging today.
- Automatic process acquisition
  - Can attach to parallel programs started as part of an MPI, OpenMP, pthread, HPF, or PVM application
  - Automatically attaches to the child process created by **fork()** or **execve**
- Can debug processes that were not started under TotalView's control
- Can temporarily add source code statements to the program you are debugging

Remote debugging

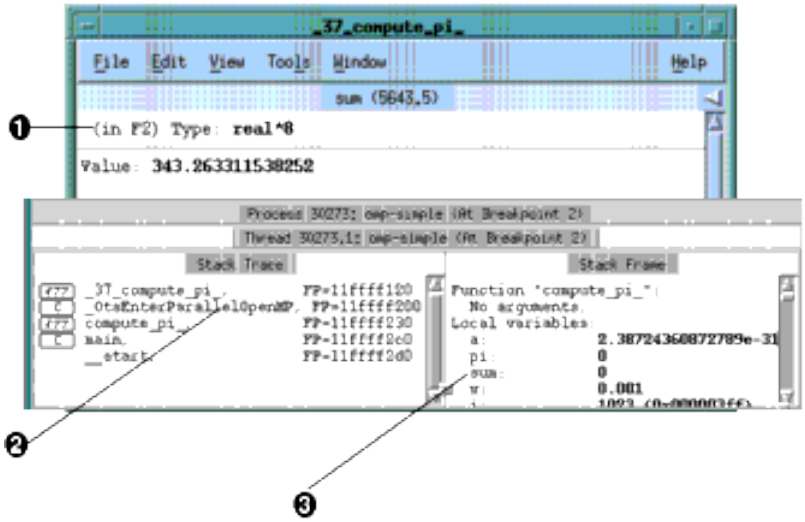
- TotalView's distributed architecture lets you debug *remote* programs over the network.



- TotalView can manage multiple remote programs and multiprocess multithreaded programs simultaneously, as shown in the following figure.



TotalView (snapshots)



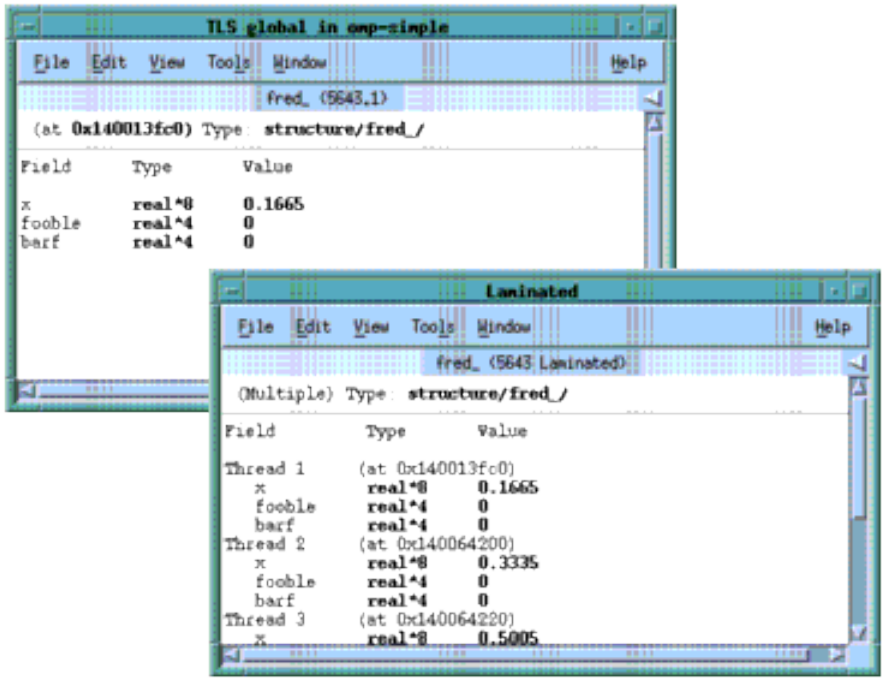
❶ OpenMP shared variables have master thread's context
❷ Original routine's stack frame selected
❸ Stack Frame Pane includes shared variables

From the on-line TotalView User's guide at www.etnus.com

SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

OpenMP threadprivate common block variables



SC2001: November 10-16, 2001

Debugging an OpenMP Code

- Source level debugging of original OpenMP code
- Access to shared and private variables
- Visibility of worker threads and stack
- Cannot step into a parallel region
  - Can place breakpoints in parallel regions

ERDC MSRC Major Shared Resource Center

KAP Pro Tools Overview

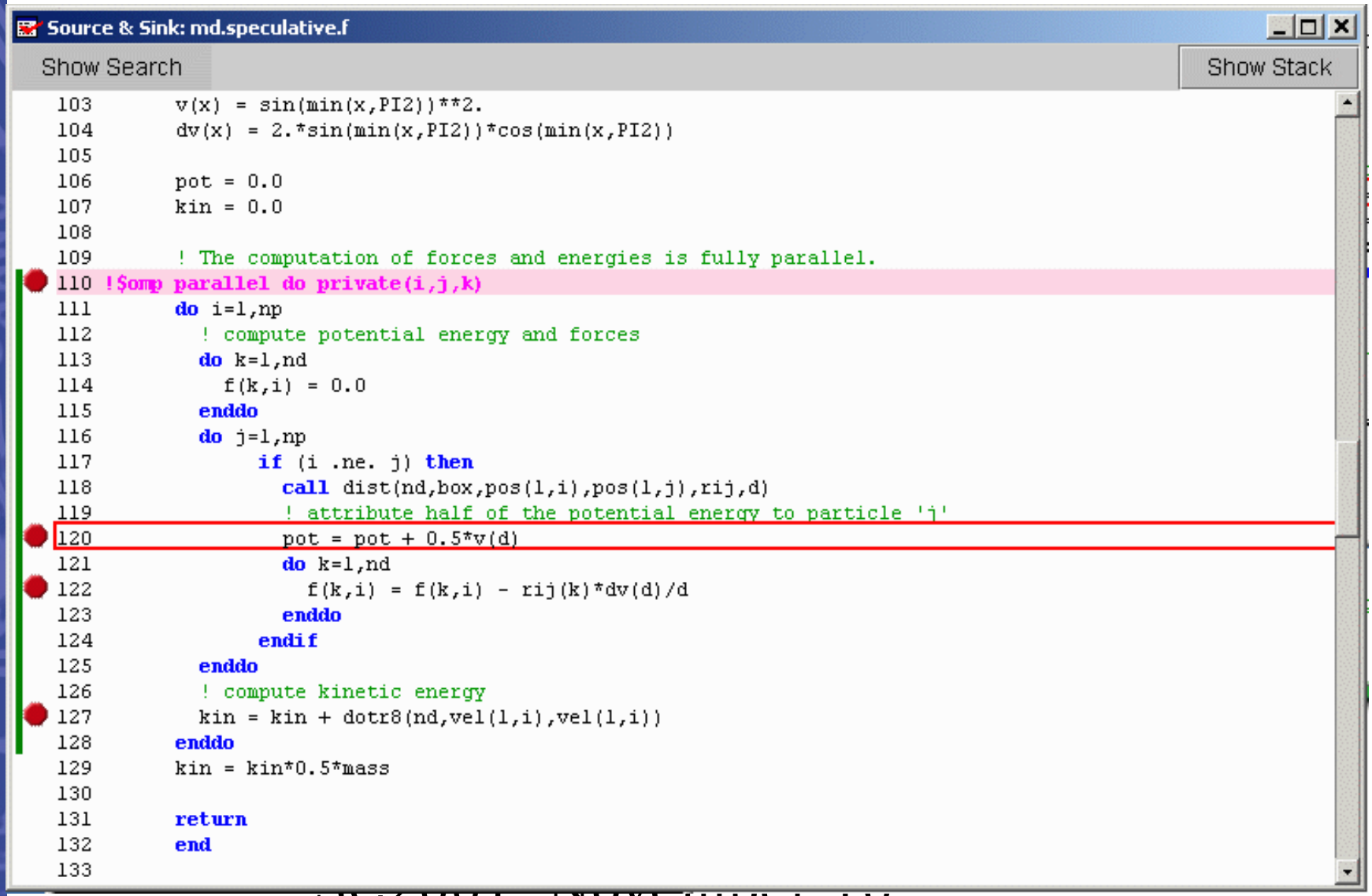
- http://www.kai.com
- Version: 4.0
  - Guide, Guideview, Assure, Assureview
- Platforms: Compaq Alpha, SGI, Sun Sparc, Intel IA32 and Itanium, HP, IBM RS/6000
- Fully compliant implementation of OpenMP standard
- Full Fortran 77/90 and C/C++ support

KAP Pro Tools

- Assure
  - tool that systematically and automatically locates parallel programming errors
  - detect race conditions, which TotalView does not do
- Assureview
  - Graphical interface to display errors found by Assure
- Guide
  - cross platform OpenMP compiler
- Guideview
  - performance analysis tool that identifies parallel performance bottlenecks and leads you to their resolution
  - graphically illustrates what each processor is doing at various levels of detail using a hierarchical summary

ERDC MSRC Major Shared Resource Center

Assureview snapshot



SC2001: November 10-16, 2001

Distributed Virtual Shared Memory

- DVSM Guide uses a single level of parallelism to distribute an application across multiple SMPs.
  - DVSM Guide is designed to port existing OpenMP programs to multiple SMPs.
  - The effort required to create a DVSM Guide program is an order of magnitude smaller.
  - MPI+OpenMP programs, because of the large effort expended to create them, tend to scale better than DVSM OpenMP programs.
  - It is KAI's belief that if the same effort is expended toward both models, the scalability should easily be the same. The difference is that DVSM Guide will let you get reasonable performance with small effort.
- KAI suggests not to use DVSM on the newer hardware that performs the same techniques.

ERDC MSRC Major Shared Resource Center

Vampir/Vampirtrace

- http://www.pallas.com
- Version: 2.5 (Vampir), 2.0 (Vampirtrace)
- Platforms
  - Compaq, Sun, IBM, SGI, HP, x86
- Support for
  - MPI
  - C/C++ and Fortran

Vampir/Vampirtrace

- What can it show you?
  - Cache misses
  - TLB misses
  - Load balance
  - Debugging
  - Where to spend your time

ERDC MSRC Major Shared Resource Center

Vampir 2.5

- New features
  - MPI collective operations
  - Optional MPI-IO and Cray SHMEM support
  - New displays
- Coming
  - Interaction with KAP Pro Tool Set and PAPI
  - And TotalView front-end

SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

VAMPIRTRACE

- VampirTrace Compile
  - CRAY T3E
    - f90 -I/usr/local/applic/VAMPIRTRACE/v2.0/include -c file.F -DUSE_VT
    - f90 -o <executable> file.o -L/usr/local/applic/VAMPIRTRACE/v2.0/lib –lVT -lpm
  - IBM
    - mpxlf90 -I/gpfs/cots/VAMPIRTRACE/include -c file.F -WF,-DUSE_VT
    - mpxlf90 -o <executable> file.o -L/ gpfs/cots/VAMPIRTRACE/lib –lVT -lld
  - SGI
    - f90 -64 -I/usr/local/applic/vampirtrace2.0/include -c file.F -DUSE_VT
    - f90 -64 -o <executable> file.o -L/ usr/local/applic/vampirtrace2.0/lib –lVT –lmpi ldwarf –lelf -lm

VAMPIRTRACE INSTRUMENTATION

```
#include <stdio.h>
#include "mpi.h“
#ifdef USE_VT
#include "VT.h"
 void VT_setup(void);
/*instrumentation  */
#endif

int main(int argc, char* argv)
{
#ifdef USE_VT
  VT_traceoff();          /*  Tracing off */
#endif
  MPI_Init(&argc, &argv);
#ifdef USE_VT
  VT_setup();                 /*
Instrumentation */
  VT_traceon();               /* Tracing
on */
  VT_begin(99);               /* Main
program */
#endif
  ******code*************
#ifdef USE_VT
  VT_end(99);         /* Leave main
program */
#endif
  MPI_Finalize();
 return 0;
```

```
#include "mpi.h"
#ifdef USE_VT
#include "VT.h"
#endif
void
VT_setup(void)
     /* Instrumentation setup */
{
  /**
   * Define the instrumentation
symbols
   **/
VT_symdef(99, “Main”, “Main");
VT_symdef(100, “Exp_Sub",
"Setup");
VT_ symdef(101, “Next_sub",
“Duty");
}
```

```
#include <stdio.h>
#include "mpi.h"
#ifdef USE_VT
#include "VT.h"
 #endif
 void
Exp_Sub(******)
  int       ****;
{
#ifdef USE_VT
  VT_begin(100);
#endif
*********code*********
#ifdef USE_VT
  VT_end(100);
#endif
}
```

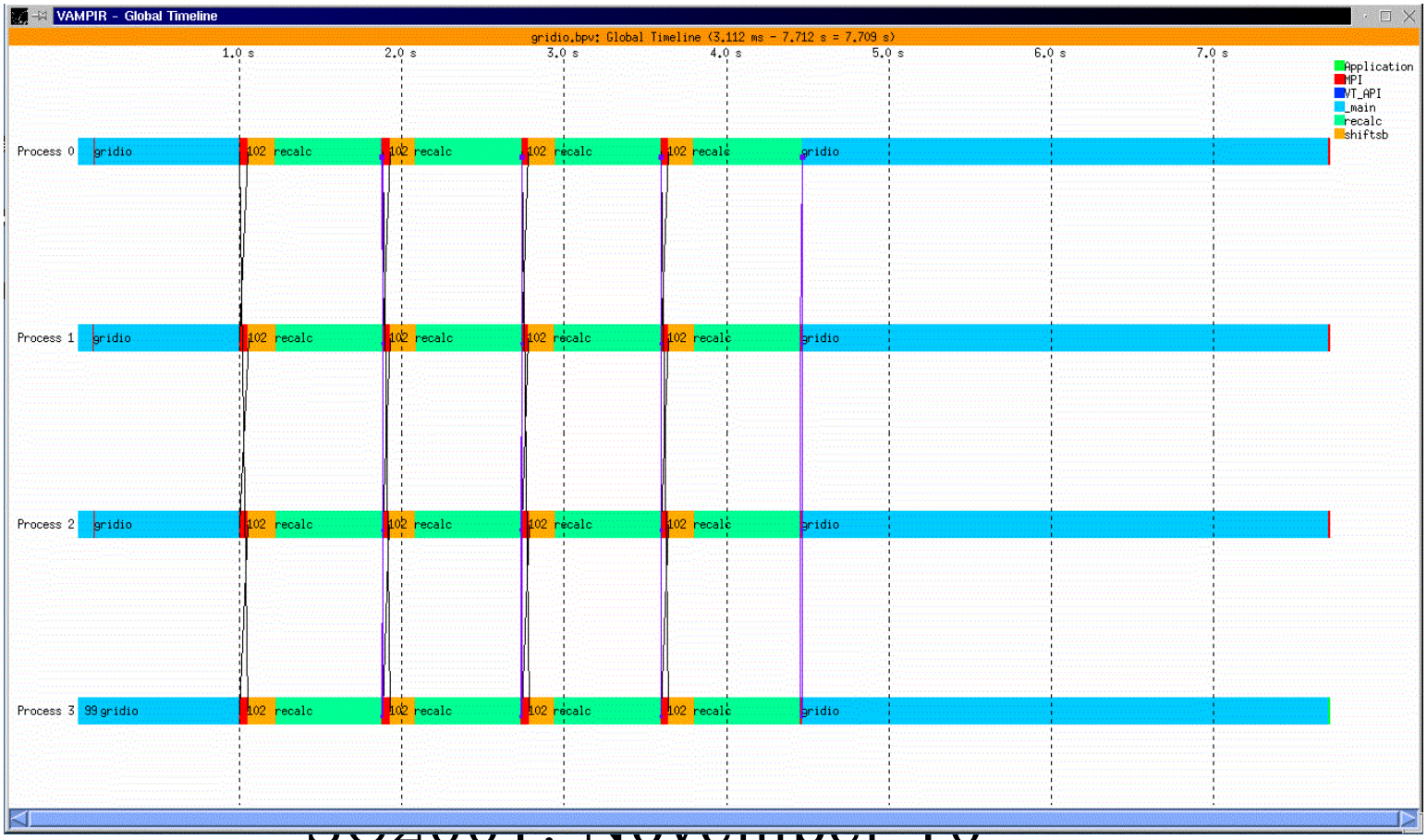
SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

VAMPIR

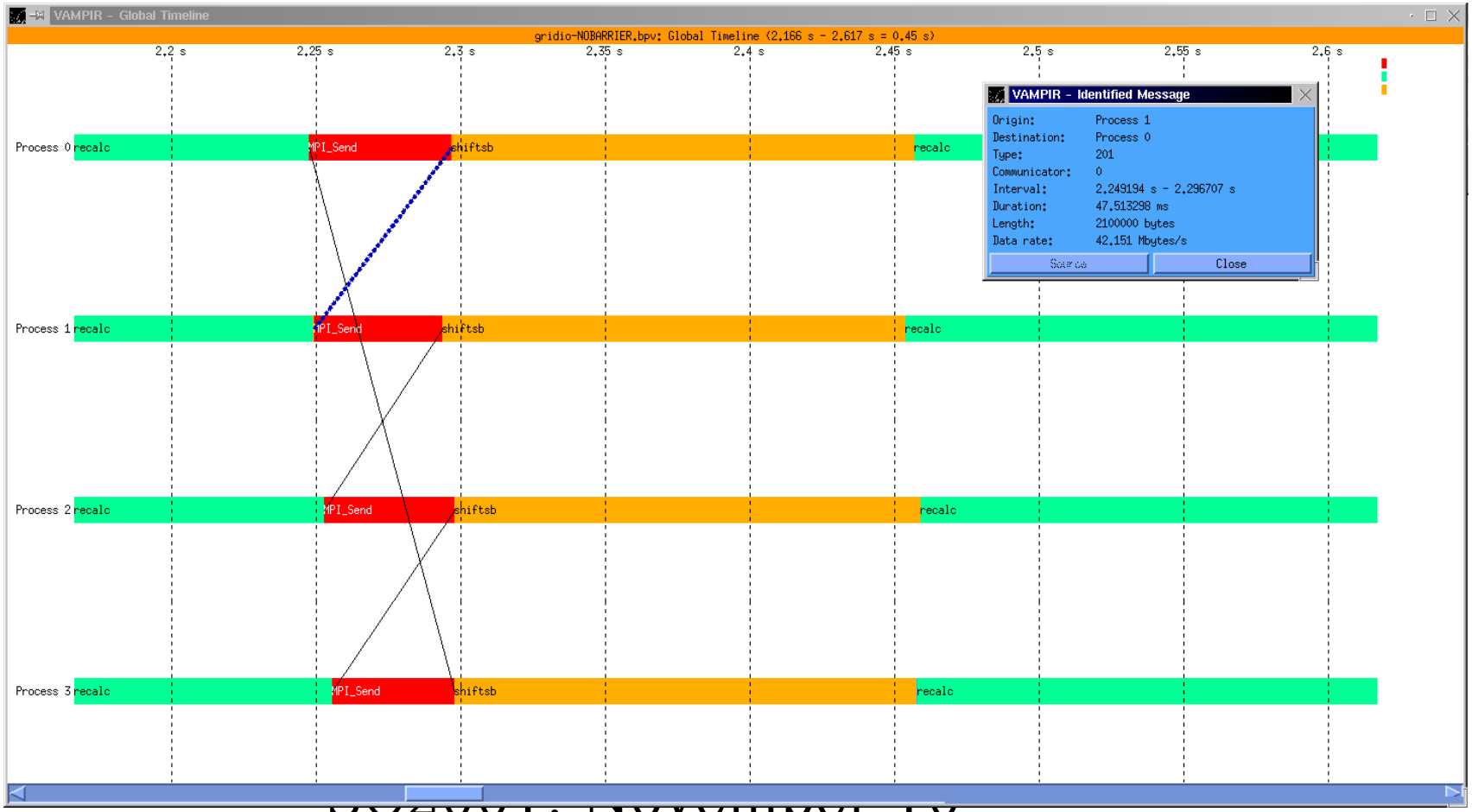
- Vampir Viewing Trace File
  - Viewing
    - setenv DISPLAY machine:0.0
    - vampir tracefile.bvp

VAMPIR – Global Timeline

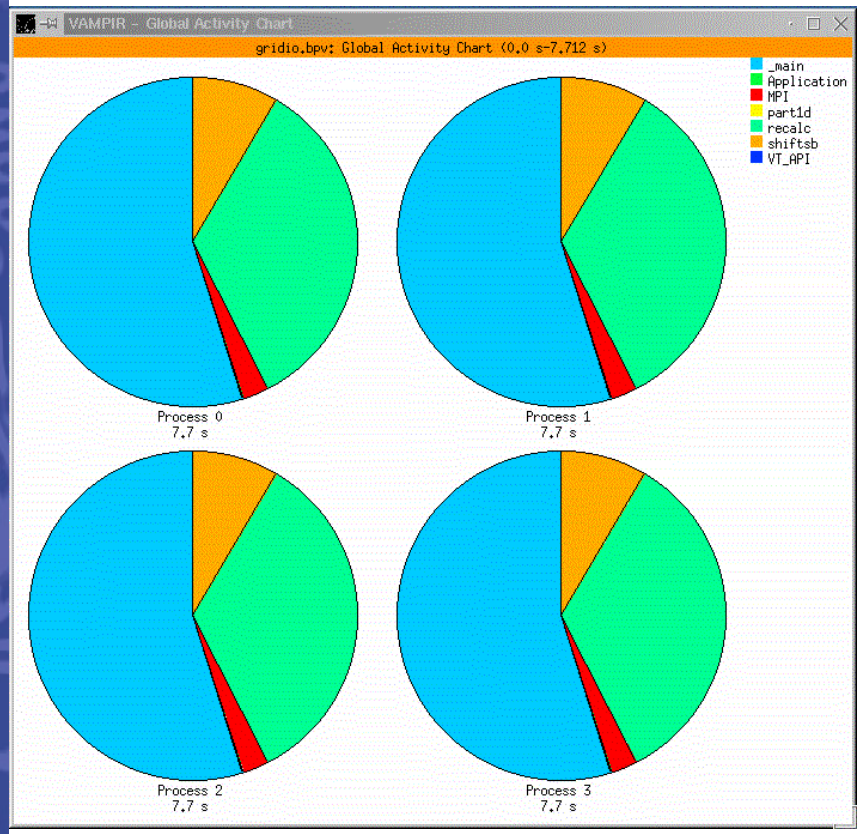


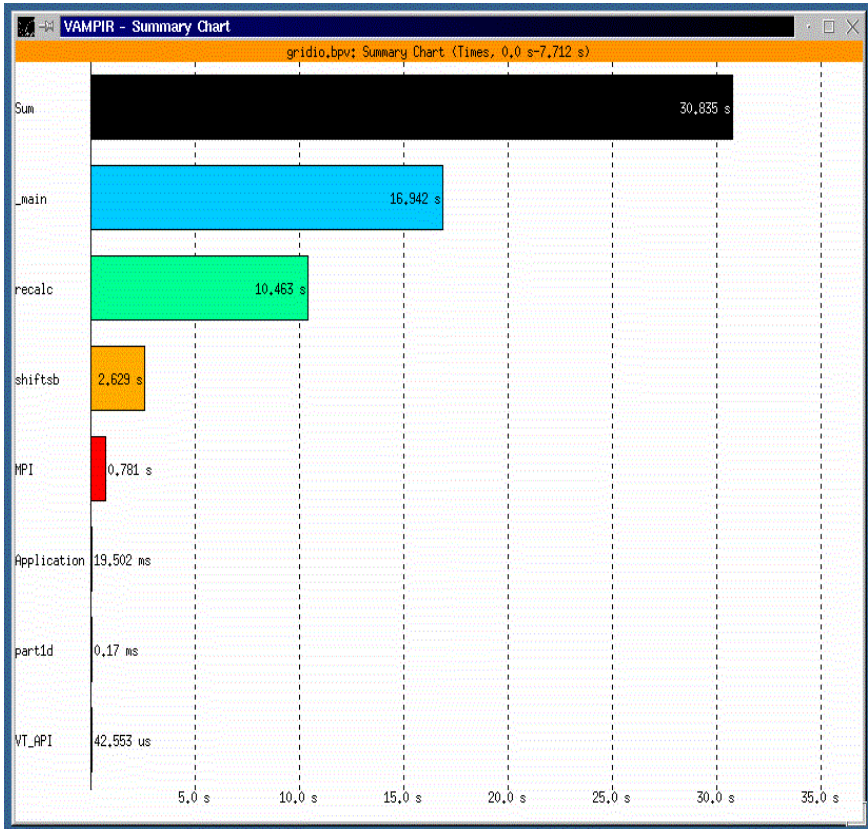
SC2001: November 10 – 16, 2001

VAMPIR – Message Information



VAMPIR – Activity, Summary





SC2001: November 10-16, 2001

Vendor-specific profilers

- Cray T3E
  - PAT (timing, cache misses, load balance)
  - MPP Apprentice
- IBM SMP/SP
  - PROF/GPROF (timing)
  - Xprofiler
- SGI O3K/O2K
  - PERFEX (L1/L2 cache, TLB, MFLOPS)
  - SPEEDSHOP (timing, memory, I/O, hardware)

Implementation of Dual-Level Parallelism

SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

Mixed-Mode Parallelism

- Some applications have two natural levels of parallelism.
- MPI performance degrades when
  - Domains become small
  - Message latency dominates computation
  - Parallelism is exhausted
- OpenMP/Pthreads typically have
  - Lower latency
  - Can maintain performance at finer granularity
- OpenMP and MPI are complementary
  - Many OpenMP threads can exist within each MPI process.
  - A single application can be built as a collection of inter-communicating multithreaded processes.

ERDC MSRC Major Shared Resource Center

Building MPI/OpenMP Programs

- Understand code/Profile
  - Are there independent components/frequencies?
    - Embarrassingly parallel
  - Identify dominate routines
  - Is a dual-level parallel model called for?
- Issues
  - Memory usage (threads and stack)
  - Function pointers in C
  - Team of threads calling MPI routines (avoid to keep simple)
  - Load balancing – automatically determine number of threads to be used for an MPI process
- Debug
  - Debug MPI and OpenMP separately first, then together

ERDC MSRC Major Shared Resource Center

Compiling MPI/OpenMP Programs

- Build code with an OpenMP compiler and link to the MPI libraries and headers.
  - IBM: use mpxlf_r, mpcc_r
  - SGI: use f90 –mp –lmpi
  - Compaq: use f90 –omp –lmpi
  - KAI compiler
    - Generally: guidef90 –lmpi or guidec –lmpi
- Make sure the compiler you use is “thread safe.”

Running MPI/OpenMP programs

- Run like ordinary MPI programs, except
  - Need to tell batch queuing system that you want more processors than MPI processes
  - May have to specify how processes get mapped to processors
  - Example
    - export OMP_NUM_THREADS=4
    - mpirun –np 8 a.out
    - Ask for 32 processors on the parallel machine
  - The batch system may have no way to deal with threads.

ERDC MSRC Major Shared Resource Center

Tip

- Make sure all MPI calls are thread safe.
  - Make MPI calls outside of OpenMP parallel regions.
    - Obviates need for thread-compliant MPI library
    - Performance benefit, only master does communication
      - Less network congestion
  - Within a parallel region, MPI communication should be limited to a single thread.
    - Avoids multiple or redundant MPI calls
    - Possibly position communicating thread on a processor with best access to network

Debugging/Tuning

- Debug with TotalView
- Tune with Vampir and/or Guide
- There are vendor-supplied alternatives to TotalView and Vampir.

Application Examples: Lessons Learned

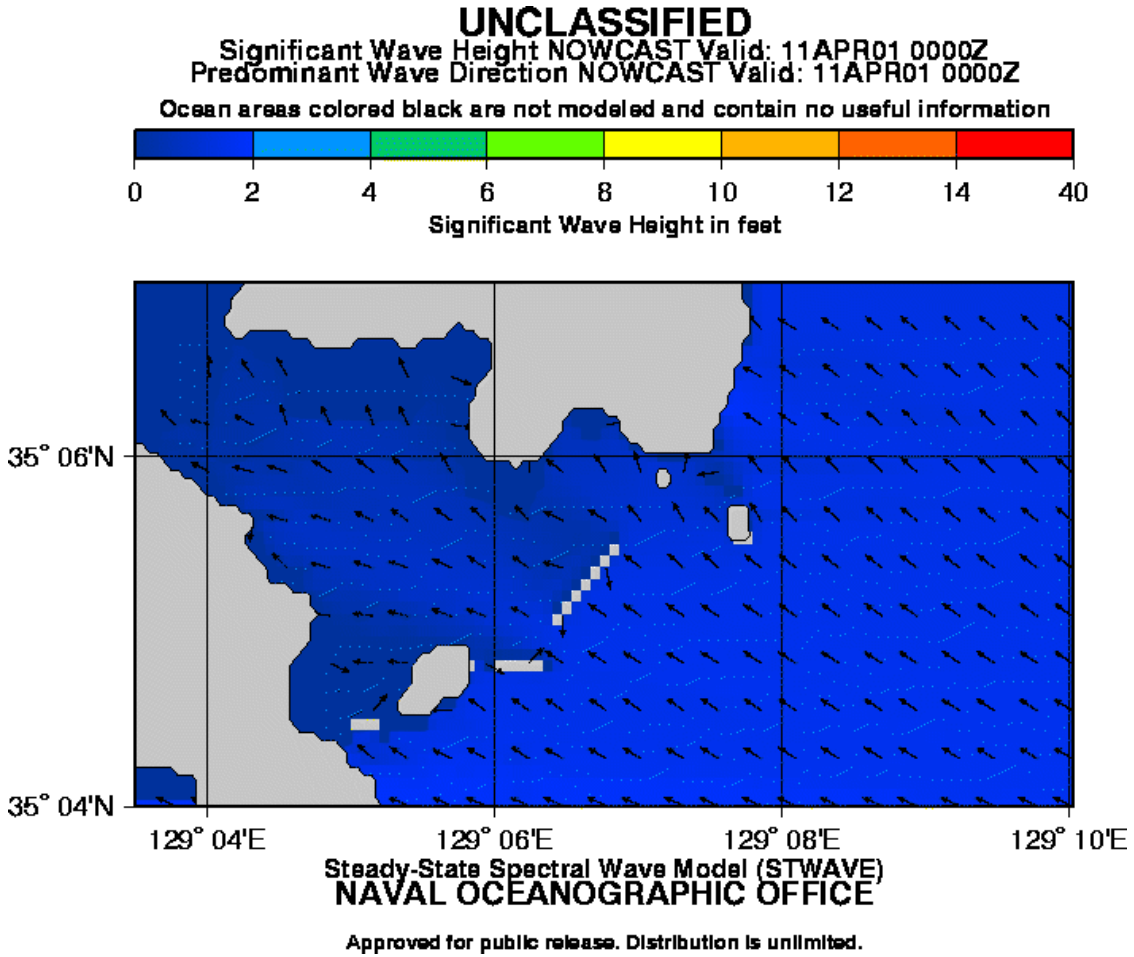
SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

STWAVE

- Routinely used on coastal projects by the Corps of Engineers to estimate wave heights, periods, and directions.
- Often uses input or supplies input to other codes such as WAM, COAMPS, or SURF.
- *STWAVE: A Case-Study in Dual-Level Parallelism*, Rebecca Fahey and Jane Smith, ERDC MSRC PET Technical Report No. 01-28, July 2001.

STWAVE



SC2001: November 10-16, 2001

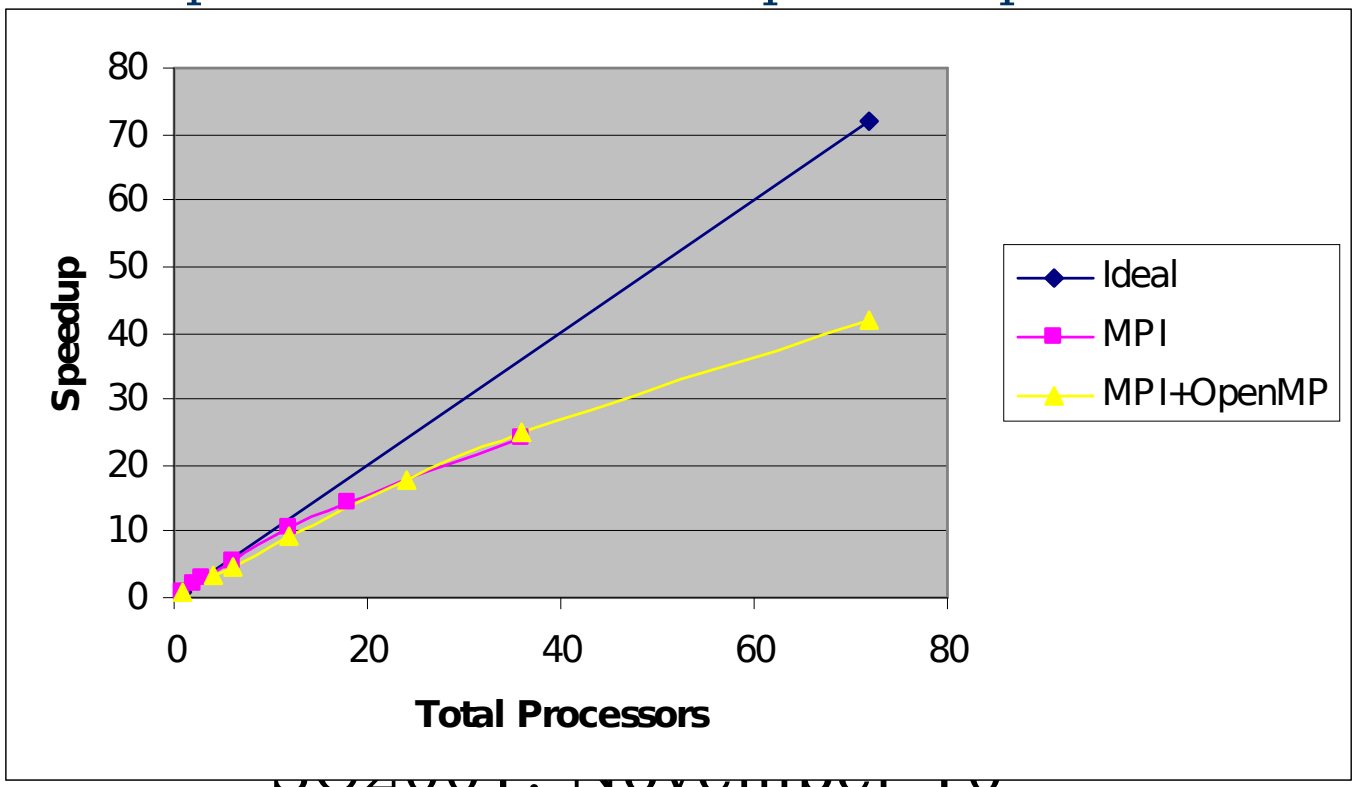
STWAVE Parallelization

- MPI
  - Distribute each independent wave run to the available processors.
  - Each frequency does the same amount of work so a static distribution of frequencies is used.
  - All bathymetry and model parameter data are replicated for each processor.
- OpenMP
  - Exploit loop-level parallelism.
  - Saves MPI communication overhead.
  - OpenMP directives amounted to about 50 lines of additional code.
  - Portability was maintained.

ERDC MSRC Major Shared Resource Center

STWAVE Speedups

ERDC IBM Power3 SMP (8-way)
Run: 36 frequencies
Hence, MPI will not scale beyond 36 processors.
2 OpenMP threads used per MPI process.



SC2001: November 10-16, 2001

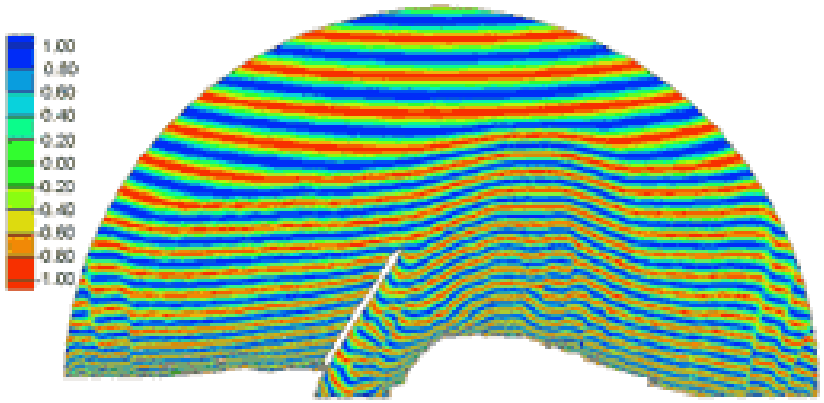
CGWAVE

- Used to predict wave climate in the near-shore area for military and civil engineering activities.
- For example, CGWAVE is used to simulate the response of a harbor's surface caused by incoming ocean waves.
- Being developed by the Office of Naval Research, U.S. Army Corps of Engineers, and NOAA.
- *Dual-Level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP,* Steve Bova, et al., *The International Journal of High Performance Computing Applications,* Volume 14, No. 1, Spring 2000.

SC2001: November 10-16, 2001

CGWAVE: Ponce Inlet

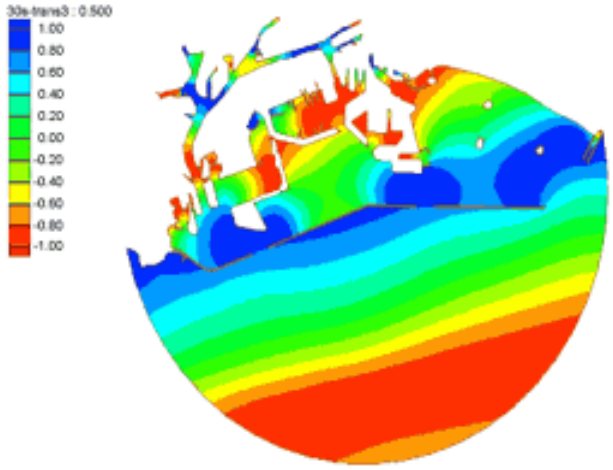
Simulation of waves in Ponce Inlet. Phase diagram. Normal incident wave angle, 15-second wave period. Coastline length is approximately 4.8 km.



SC2001: November 10-16, 2001

CGWAVE: Los Angeles Harbor

Wave penetration in Los Angeles/Long Beach Harbors
Modeled Phase Diagram.
Boundary Reflectivity set to zero; 30 s. long wave resonance study. Waves incident from bottom right.



SC2001: November 10-16, 2001

CGWAVE Parallelization

- MPI
  - Distribute each independent wave run to the available processors.
  - Each frequency does a varying amount of work so a boss-worker model is used.
  - All bathymetry and model parameter data are replicated for each processor.
- OpenMP
  - Exploit loop-level parallelism.
  - Saves MPI communication overhead.
  - Portability was maintained.
- Test Case
  - Incident waves approaching a cylindrical island.
  - 75 incident wave components
  - 50k elements
  - Timings taken on an Origin 2000 (195 MHz)

CGWAVE Boss-Worker Code

MPI Boss:

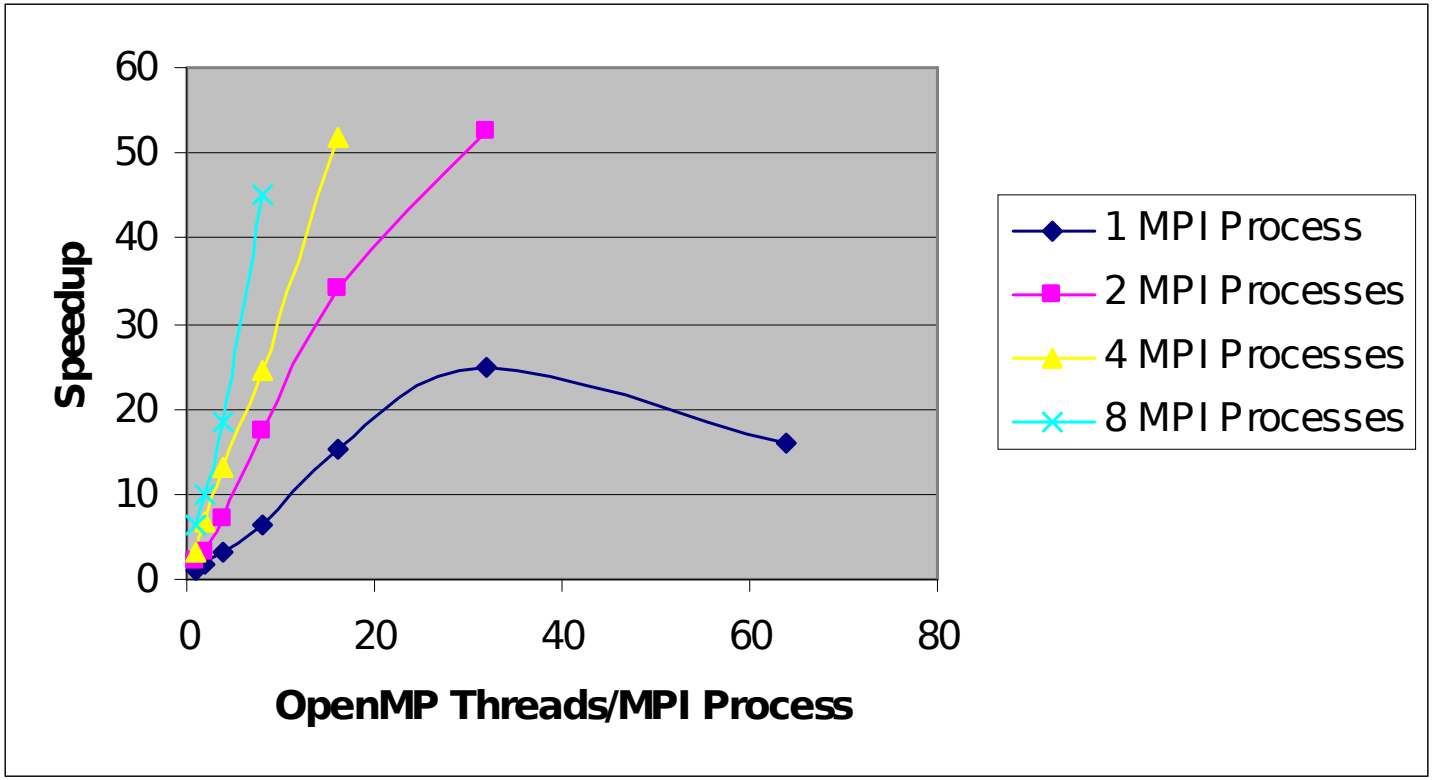
```
...Initialize MPI ...

do I = 1 to
number_of_wave_components
   blocking receive ! wait for work
request
   blocking send ! send wave
component
end do
! All wave components solved
do worker = 1 to nprocs-1
   blocking receive ! wait for work
request
   blocking send ! terminate
worker
end do
```

MPI Worker:

```
...Initialize MPI ...

do forever
   blocking receive ! ask boss for wo
   blocking send ! get wave compon
   if no_termination_signal then
      perform calculations
   else
      exit infinite loop
   endif
end do

...Finalize MPI ...
```


ERDC MSRC Major Shared Resource Center

CGWAVE Speedups

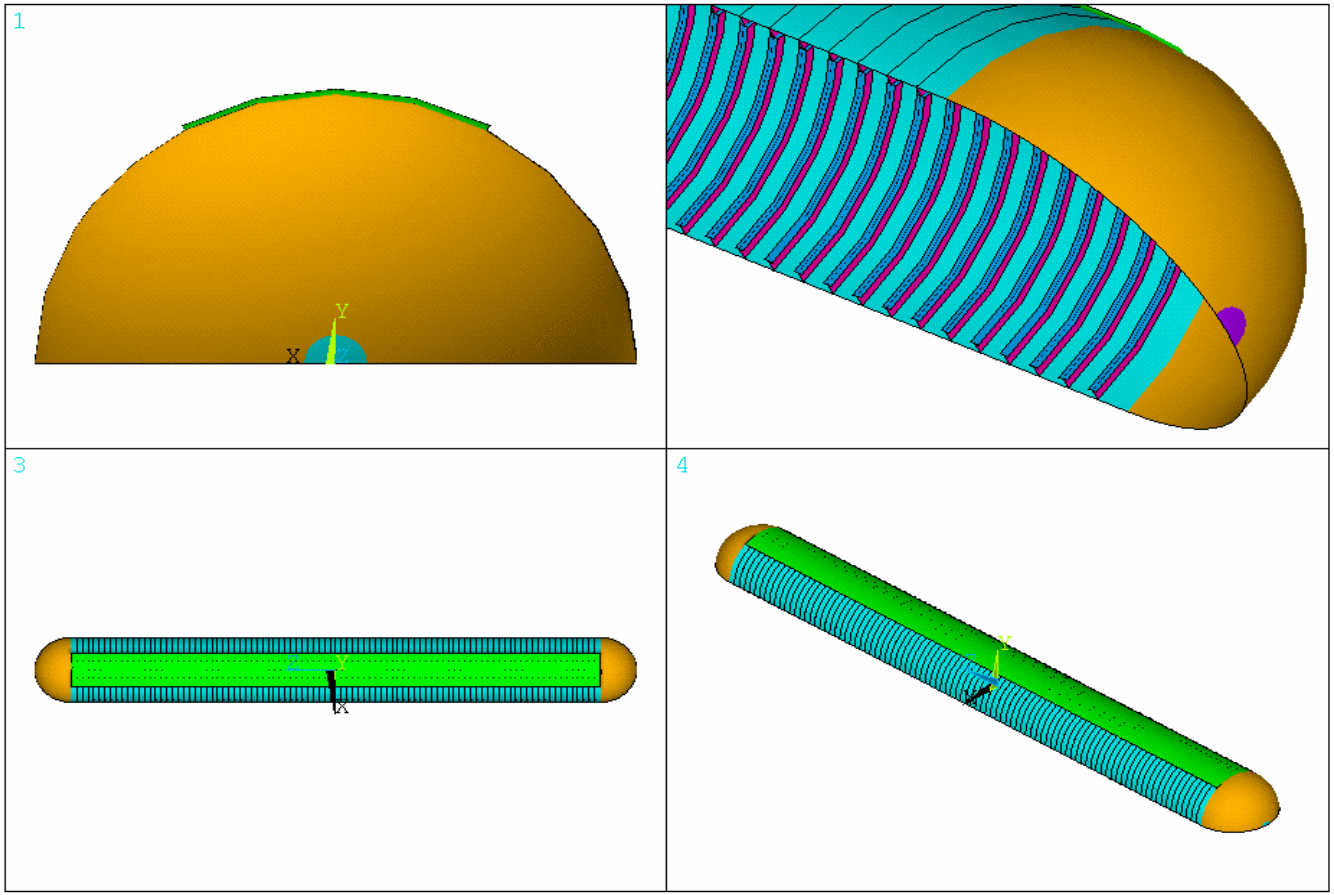


SC2001: November 10-16, 2001

SARA-3D (Structural Acoustics)

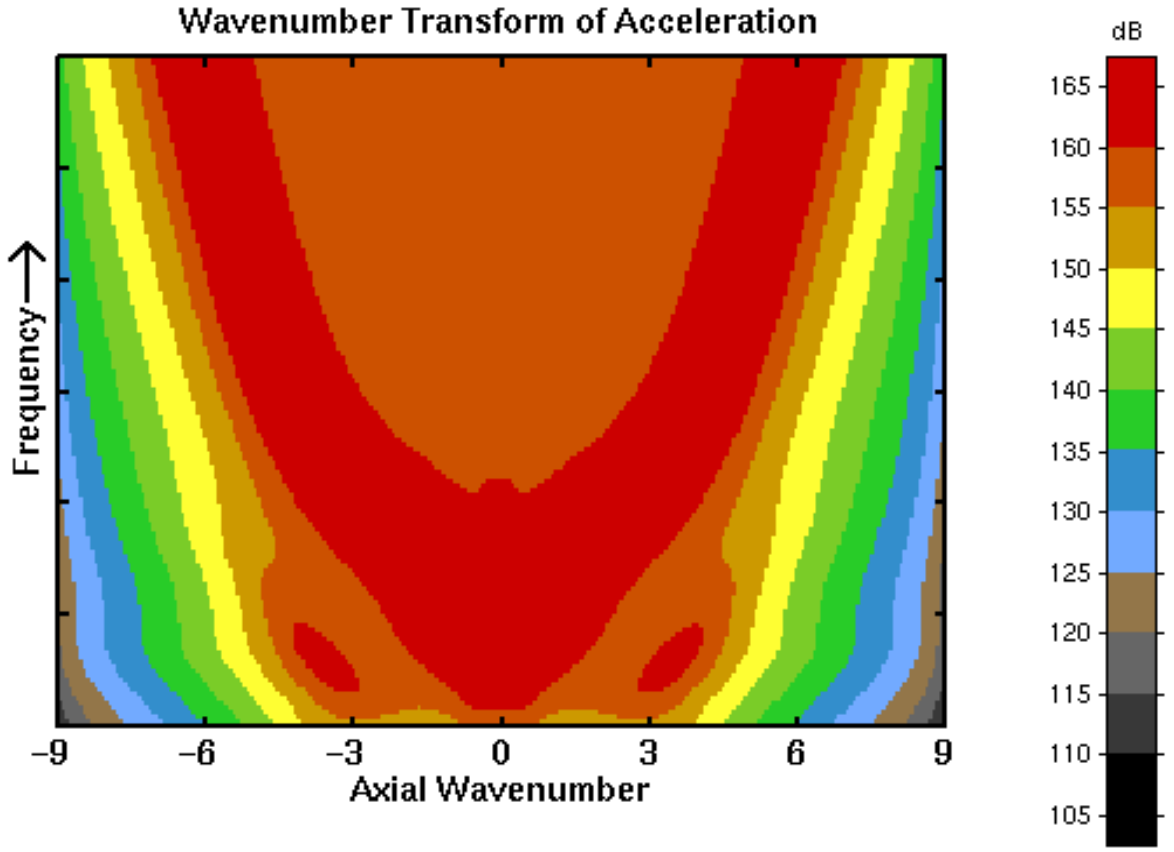
- Used primarily by DoD labs for solving problems in
  - Structural vibrations and acoustics
  - Radiation and Scattering
  - Electroelasticity
- Uses infinite elements to model the far field fluid and various types of finite elements to model the near field fluid and the structure
- This approach is more efficient than the boundary element method, which results in large dense matrix problems to solve.
- A typical use is the design of submarine hulls to reduce their sonar profiles.
- More information
  - http://www.gte.com/aboutgte/gto/bbnt/apts/offerings/sara.html

Finite Element Model for SARA-3D



SC2001: November 10-16, 2001

Typical SARA-3D Results



SC2001: November 10-16, 2001

Good Candidate for Multilevel

- Loop over incident frequencies
  - Each frequency is independent and can be done in parallel.
  - Each frequency resulted in a rather large linear system of equations to be solved using a frontal solver (Gaussian Elimination).
  - Profiles using SpeedShop on the Origin and Xprofiler on the IBM showed each set of equations resulted in a large amount of calculations in just a few areas of the code.
- Approach to Multilevel Parallelism
  - Parallelize the incident frequency loop using MPI.
  - Employ a boss-worker algorithm to achieve dynamic load balancing.
  - Use OpenMP within each frequency to further parallelize certain tasks, such as
    - element calculations
    - frontal solver

OpenMP Efficiency: Prefront Routine

- Summary of the speedup of the *prefront subroutine only* for various sizes of the sample8 test. All times are in seconds.

Problem Size (P2)	Serial	Parallel 8 OpenMP Threads	Speedup
8	570	107	5.3
12	1233	269	4.6
20	3518	569	6.2

OpenMP Efficiency: Frontal Solver

- Summary of the speedups of the *frontal solver only* for various test cases. All times are in seconds.

Test Case	Serial	Parallel 8 OpenMP Threads	Speedup
pfish (1)	4000	650	6.2
pfish (2)	2063	673	3.1
pfish (3)	1910	584	3.3
run_b (1)	6115	1494	4.1
run_b (2)	5416	1371	4.0

ERDC MSRC Major Shared Resource Center

Threaded Efficiency: Large Problem

	1 thread	2 threads	4 threads	8 threads
Front opt	67.95	37.61	21.97	16.68
Setup	262.48	183.22	143.60	143.57
Freq Sweep	4405.32	2423.11	1447.73	1010.25
Prefront	3306.53	1663.17	848.22	475.25
Solution	1055.59	735.41	583.53	522.77
Forward sol	750.46	456.00	311.58	238.64
Contours	16.40	8.24	4.42	5.00
Total time	4684.64	2615.01	1596.20	1159.31

ERDC MSRC Major Shared Resource Center

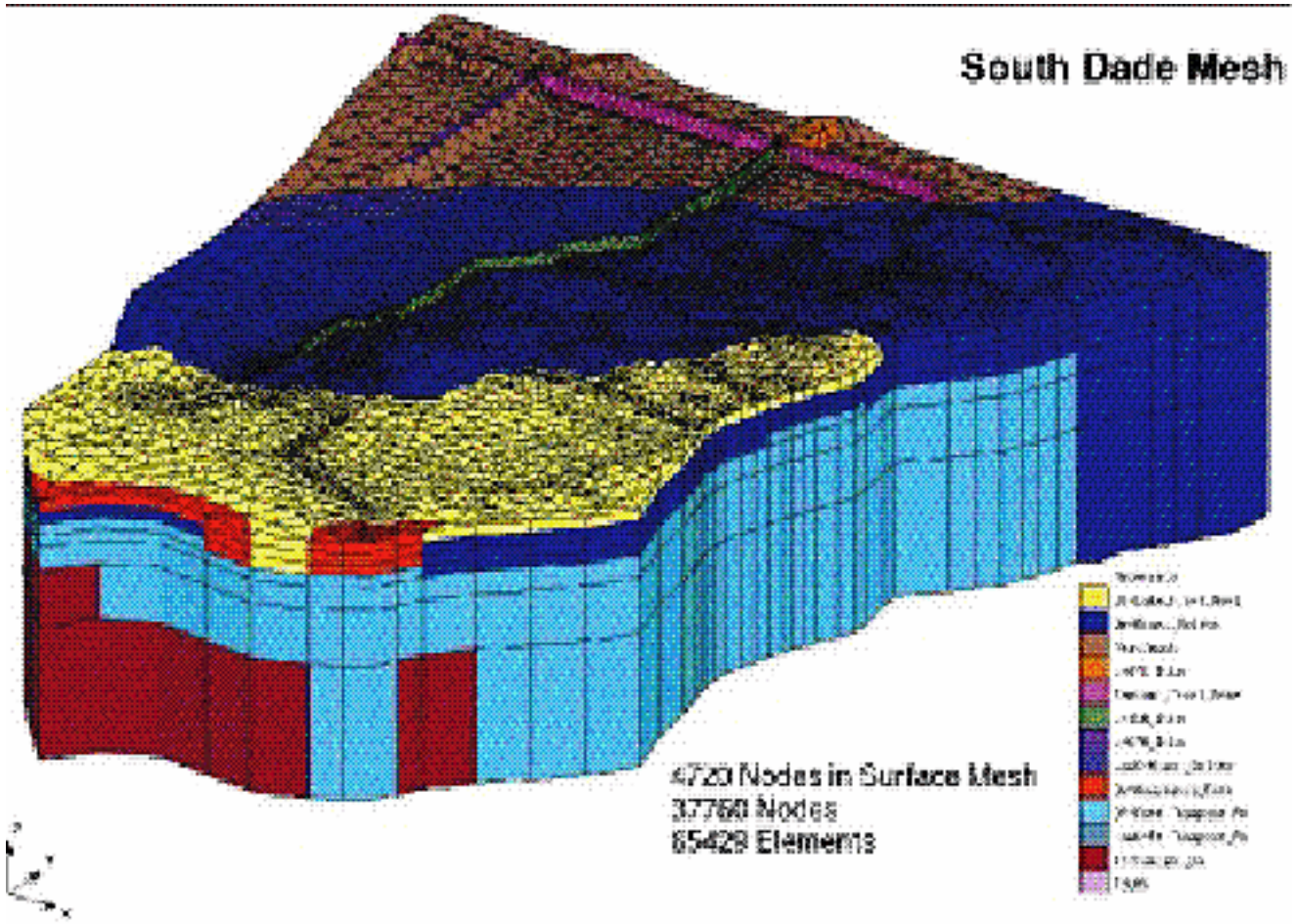
Overall Success with SARA-3D

- One model had 292,693 DOF with a frontal width of 642.
- Single Processor Origin 2000 processor took 76.5 hours.
- ERDC IBM Power3 SMP: 8-way
  - Nodes: 7
  - MPI processes per node: 2
  - OpenMP threads per MPI process: 4
  - Total number of frequencies: 48
  - Max. # of frequencies per MPI process: 4
  - Time in prefront: 0.61 hours
  - Total time: 2.54 hours
- Efficiency is not that good, but the time to solution is great.

FEMWATER123

- Models ground water flow and transport
- Parallelized using ParMetis
- Piecewise iterative solver is computational intensive
- Code spends most of its time (up to 70 percent) in one routine
- Good candidate for breaking up the work in this routine with threads

FEMWATER123: South Dade County



302001. November 10-
16, 2001

FEMWATER123: Bad News

- The one routine was called many, many times.
- Hence, the amount of work within a single call was relatively small.
- This work was not enough to amortize the time it takes to create and destroy the threads.
- Tried several ways to speed up, including fewer numbers of MPI processes with threads.
- No success.

Creating a Multilevel Code

- Start at the coarse level
  - Decide how to use MPI to break up the largest blocks of work, e.g., domain decomposition or by tasks.
  - Decide on a strategy to obtain the best possible load balance, e.g., boss-worker, static scheduling, or domain decomposition.
- Move to the fine level
  - Use a profiler to show what areas of the code are the most computationally intense
    - Xprofiler on the IBM is excellent.
    - SpeedShop on the SGI is great.
  - Explore using threads to further parallelize these areas.
  - Measure the efficiency in order to determine the optimal number of threads to use.

Lessons Learned (1)

- Program incrementally
  - Harder to do with MPI and Pthreads
  - Easier to do with OpenMP
- Program portability
  - Adhere to using those functions supported by the MPI, OpenMP, and Pthreads standards.
  - Even computers without OpenMP can still run the MPI version since OpenMP directives are actually comments.
- Use threads within MPI
  - Rule of thumb: You can use MPI within OpenMP if you are careful.
  - Easier
  - Faster

ERDC MSRC Major Shared Resource Center

Lessons Learned (2)

- Treat the threads like graduate students
  - Make them do a lot of work.
  - There is a minimum amount of work that needs to be done in order to make the overhead worth it.
- Check for correctness
  - Always have a way to check for correctness and be able to move back to a previous version of the code.
- Benefits outweigh the costs
  - The efficiency may not be the best, but the ease at which a quicker solution can be obtained is undeniable.
  - Be a scientist, not a computer scientist (unless you want to).

Future of Multilevel Parallelism

- Distributed OpenMP
  - Should be out later this year
  - Kuck and Associates will probably be the first vendor to supply this.
- MLP: Fork processes each spawning threads
  - James Taft (NASA Ames), et al., has parallelized several codes.
  - Mark Fahey (ERDC/ORNL) has explored MLP in a version of the Overflow code.
  - Positive results have been seen.
- MPI with Shmem
  - Guy Robinson (ARSC) and Fred Tracy (ERDC)
- MPI-II
  - Dynamic process creation.
  - Perhaps only one vendor has implemented this

Conclusions

- Shared memory is becoming almost a standard.
- This will always leave the possibility for a Multilevel Parallel code to be used.
- Given an existing MPI code, it is fairly easy to include OpenMP directives to provide a shorter time to solution.
- It is possible to get a better speedup with a combination of parallel methods.
- All depends on the problem

Laplace’s Equation Example

SC2001: November 10-16, 2001

Laplace's Equation

Problem: Solve Laplace's equation with a constant boundary condition.

$$\nabla^2 u = 0$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$$

SC2001: November 10-16, 2001

ERDC MSRC Major Shared Resource Center

Finite Difference Approximation

- Assume the following:
  - One-dimension (to make things easier)
  - Regular grid in the x direction

$$\frac{\partial^2 u}{\partial x^2} = 0$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{(\Delta x^2)} + O(\Delta x^2) = 0$$

More Finite Difference Approximation

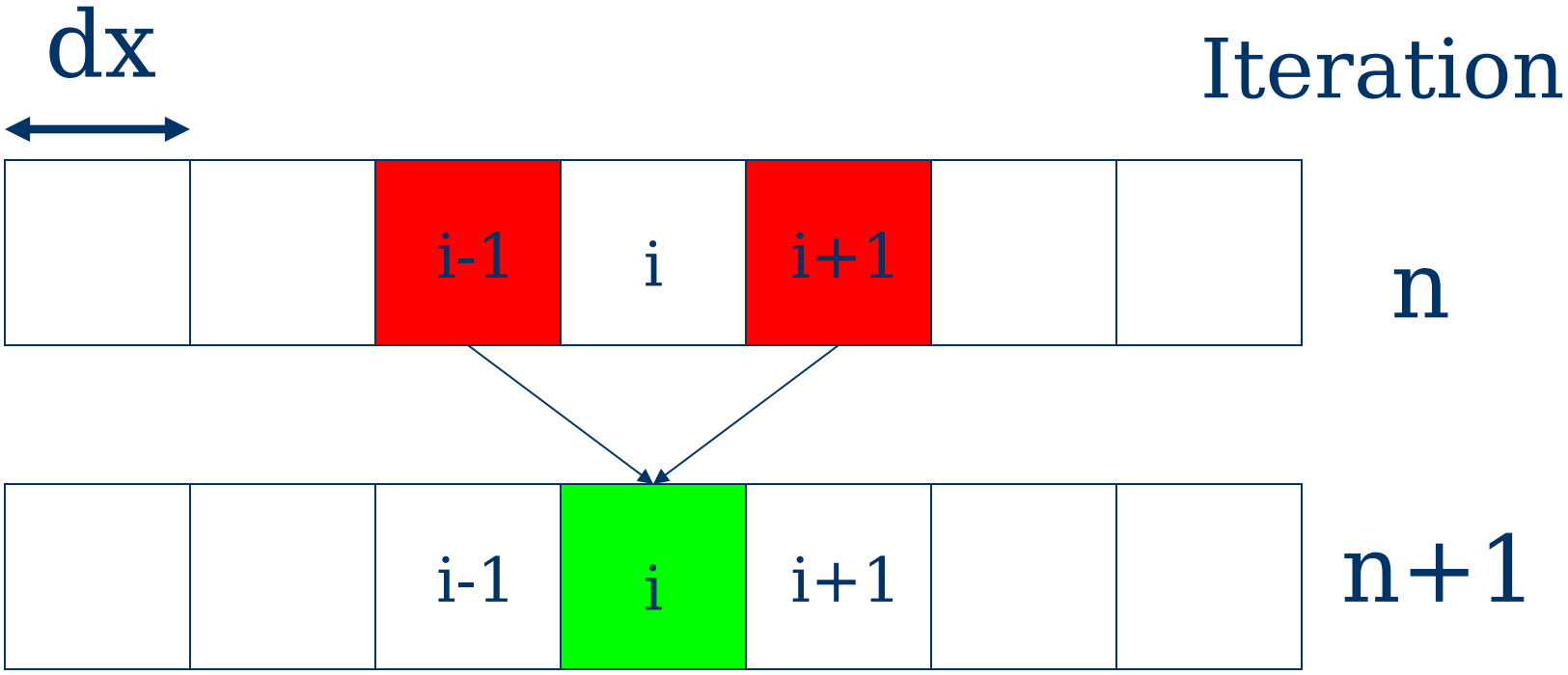
- Solve for $u(x_i)$

$$u(x_i) = \frac{1}{2}\left[u(x_{i+1}) + u(x_{i-1})\right]$$

- Use the solution at iteration n to generate the values at the next iteration n+1

$$u^{n+1}(x_i) = \frac{1}{2}\left[u^n(x_{i+1}) + u^n(x_{i-1})\right]$$

1D Domain



Initial Conditions

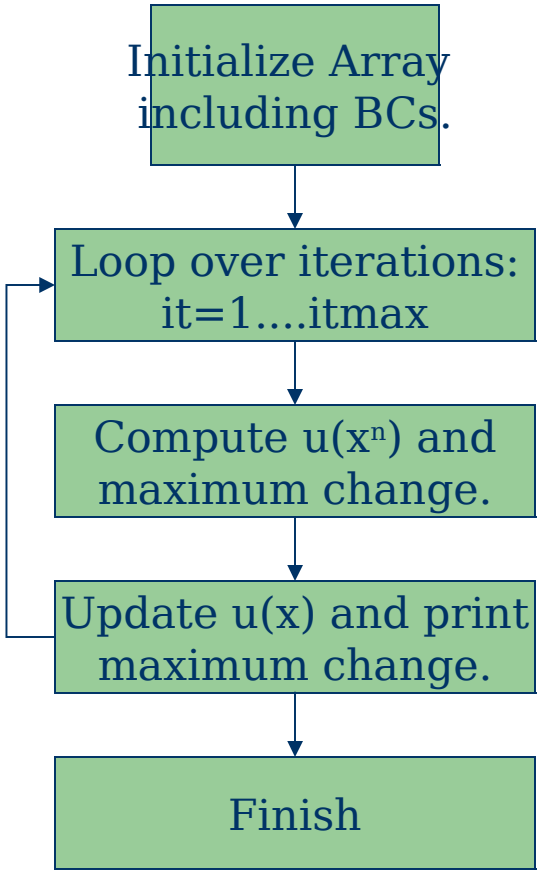
Set all the initial values to zero except for the edge, which is set to some constant, C.

$$u(x_i) = 0$$

$$u(x_L) = C$$

ERDC MSRC Major Shared Resource Center

Serial Algorithm


Initialize Array including BCs.
Loop over iterations: it=1….itmax
Compute u(x^n) and maximum change.
Update u(x) and print maximum change.
Finish

Serial Code

```
#include <stdio.h>
#include <math.h>

int main (int argc, char *argv[]) {
  const imax  = 20;      /* Number of points.
        */
  const itmax = 10;      /* Number of
iterations, n.  */
  const umax  = 10.0;   /* Boundary
Condition.      */

 int i, j, it;
  double u[imax];       /* Array of elements.
      */
  double du[imax];      /* Array of shifts to u.
     */
  double d,dumax;

  /* Initialize the entire array to zero and
include
  the boundary condition. */
  for (i=0; i<imax; i++){
    u[i]  = 0.0;
    du[i] = 0.0;
  }
```

Initialize the arrays.

Boundary Condition

ERDC MSRC Major Shared Resource Center

Serial Code (Continued)

```
  /* Loop through the iterations, compute the
finite
    differences, and update the array u. */
  for (it=0; it<itmax; it++) {
    dumax = 0.0;
    for (i=1; i<imax-1; i++) {
      d     = 0.5*(u[i+1]-2.0*u[i]+u[i-1]);
      du[i] = d;

      d = d<0 ? -d:d;
      if (dumax < d) dumax = d;
    } /* end of I loop */

    printf("it = %d  dumax = %12.8f \n",
it,dumax);

    for (i=1; i<imax-1; i++)
                u[i] += du[i];
   } /* end of it loop */

   for (i=0; i<=imax-1; i++)
     printf("i=%3d  u[i]=%12.8f \n",i,u[i]);
  return 0;
}
```

Loop over iterations.

Compute change in u.

Find the maximum change and print.

Update u.

Print the final u values.

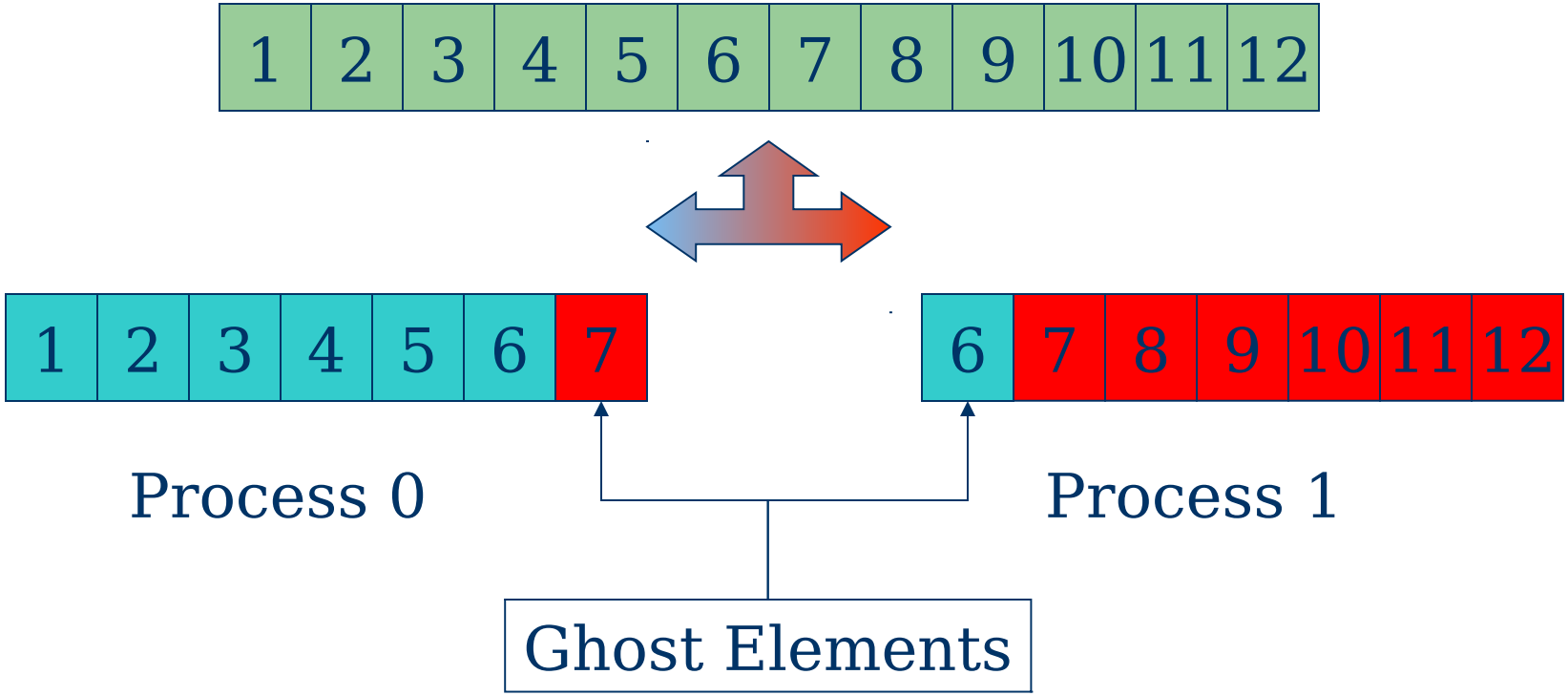
How can MPI be used?

- What issues need to be considered when using MPI to create a parallel Laplace equation solver?
  - *What tasks can be performed in parallel?* Loop over the iterations contains a data dependency that cannot be parallelized. However, the inner loop used to compute the next iteration of u(x) can be parallelized.
  - *How do we break up the array u(x) over the processes?* Simply divide up the array as equally as possible over the number of MPI processes. This gives a good first cut at load balancing.
  - *After the nth iteration, does each process hold the correct information needed to perform its task?* No. Each process needs to know information that is held by its neighbors. Hence, at either the beginning or the end of each iteration, the most current values of the array u(x) need to be sent between neighboring processes.
  - *How do you check for correctness and efficiency?* Compare the final answers to the serial answers and use some timers to compute the codes efficiency.

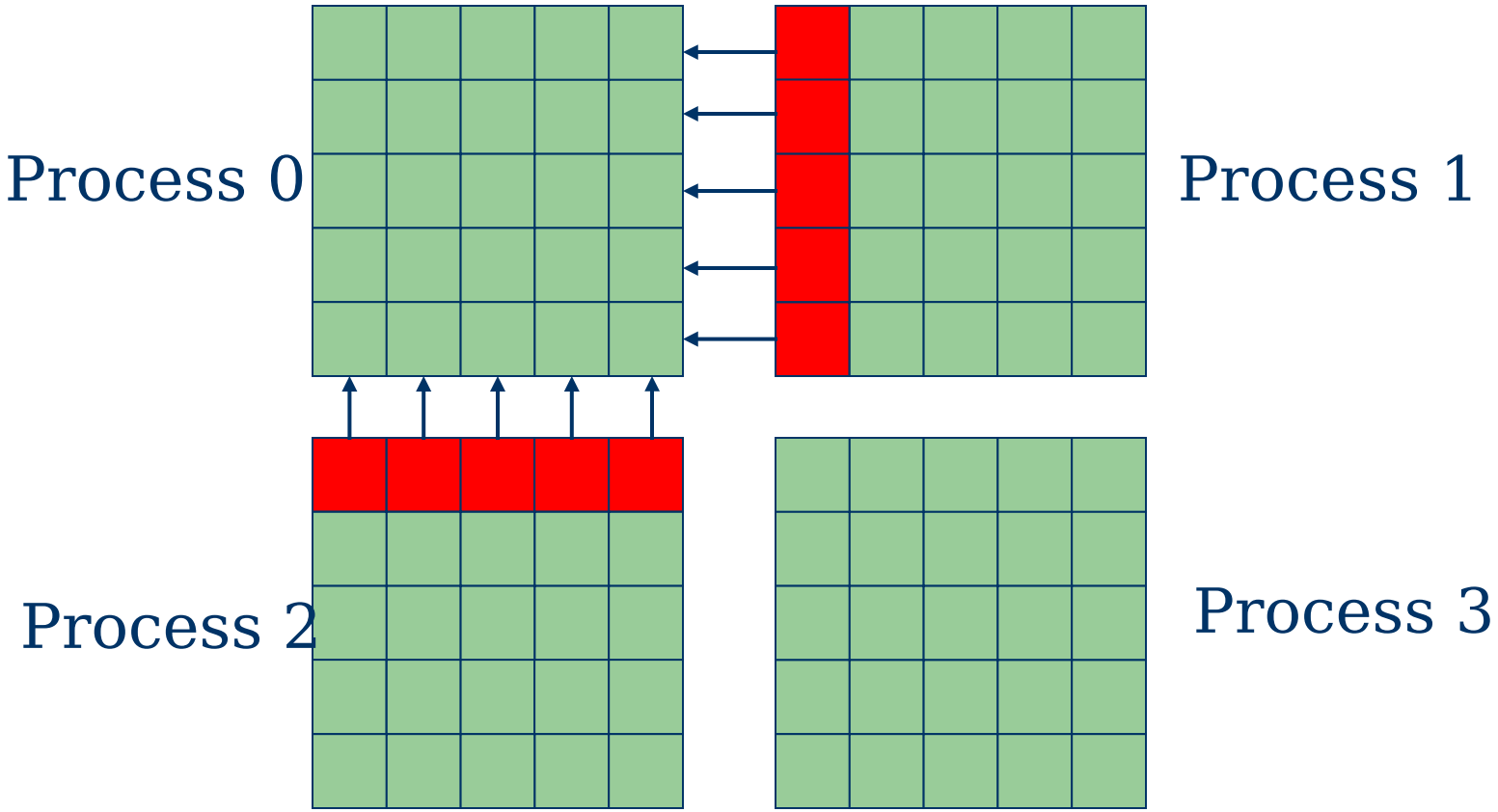
ERDC MSRC Major Shared Resource Center

Ghost Elements (Nodes): 1D

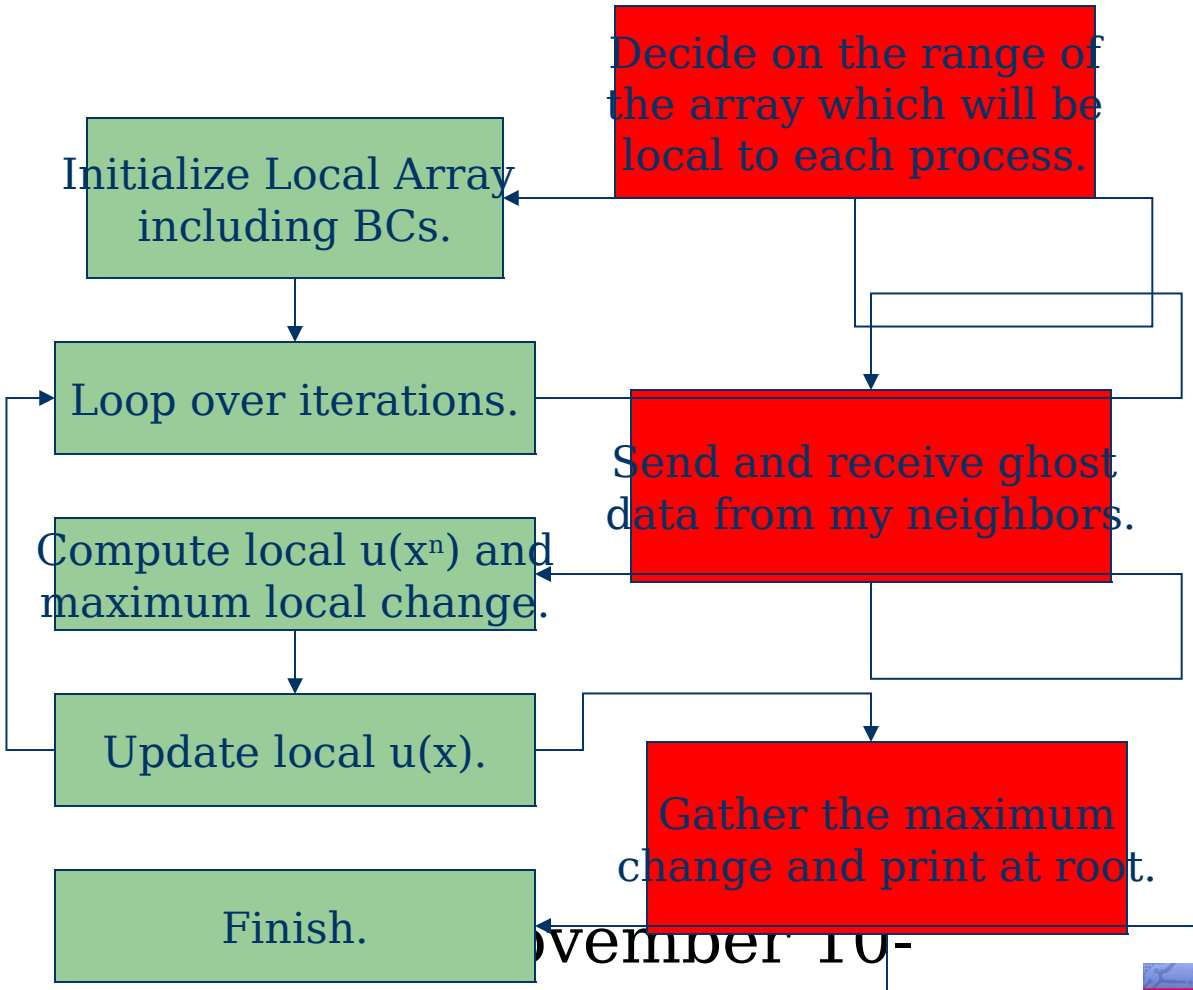
Elements that a process needs but does not own or comput



Ghost Elements (Nodes): 2D



Parallel Algorithm


Decide on the range of the array which will be local to each process.
Initialize Local Array including BCs.
Loop over iterations.
Send and receive ghost data from my neighbors.
Compute local u(x^n) and maximum local change.
Update local u(x).
Gather the maximum change and print at root.
Finish.

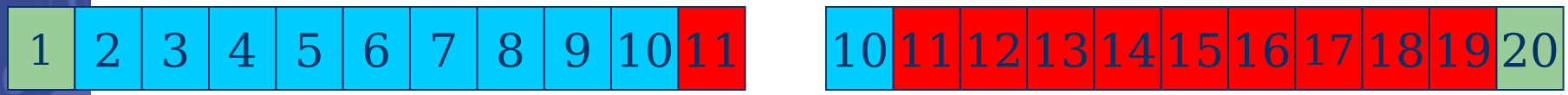
Parallel Code: Specific to 2 MPI Processes

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
  const imax  = 20;
  const itmax = 20;
  const umax  = 10.0;

  int myrank, nproc, ROOT, LAST, i, j, it;
  int imax_local, istart, iend;
  int *num_elements, num_local, total_num;
  int *displacements;
  double *u, *du, d,dumax,dumax_local;
  double sendbuf,recvbuf;

  MPI_Request request;
  MPI_Status  status;

  /*Initialize MPI.*/
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

```
  ROOT = 0;
  LAST = 1;

  /* Compute the range over which
to sum. */
  imax_local = imax/nproc + 1;
  istart = 1;
  iend   = imax_local-2;

  /* Allocate the arrays.*/
  u = (double *) malloc(imax_local*
                        sizeof(double));
  du = (double *) malloc(imax_local*
                        sizeof(double));

  /* Initialize the arrays.*/
  for (i=0; i<imax_local; i++) {
     u[i]  = 0.0;
     du[i] = 0.0;
    }

  /* Boundary Condition.*/
  if (myrank==1)
    u[imax_local-1] = umax;
```

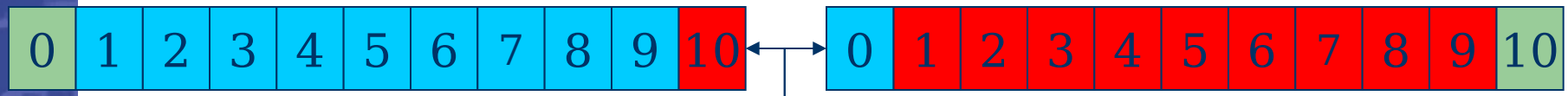
Local Arrays and Ghost Elements

Divide up the array.



Renumber locally.

imax_local imax_local



istart=1 iend=9 Ghost Elements istart=1iend=9

Parallel Code (Continued)

```
for (it=0; it<itmax; it++) {
  /*Pass the ghost elements.*/
  if (myrank==0){
    sendbuf = u[imax_local-2];
    recvbuf = 0.0;

MPI_Send(&sendbuf,1,MPI_DOUBLE,1,100,
            MPI_COMM_WORLD);

MPI_Irecv(&recvbuf,1,MPI_DOUBLE,1,100,
            MPI_COMM_WORLD,&request);
    MPI_Wait(&request,&status);
    u[imax_local-1] = recvbuf;
  }
  else if (myrank==1){
    sendbuf = u[1];
    recvbuf = 0.0;

MPI_Send(&sendbuf,1,MPI_DOUBLE,0,100,
            MPI_COMM_WORLD);

MPI_Irecv(&recvbuf,1,MPI_DOUBLE,0,100,
            MPI_COMM_WORLD,&request);
    MPI_Wait(&request,&status);
    u[0] = recvbuf;
```

Loop over maximum number of iterations.

PE 0: Send
Irecv
Wait

PE 1: Send
Irecv
Wait

Parallel Code (Continued)

```
/*Perform the local update.*/
      dumax_local = 0.0;
      for (i=istart; i<=istop; i++) {
        d     = 0.5*(u[i+1]-2.0*u[i]+u[i-1]);
        du[i] = d;

        d = d<0 ? -d:d;
        if (dumax_local < d)
          dumax_local = d;
      }
      for (i=istart; i<=istop; i++)
        u[i] += du[i];

 /*Find the maximum value of the change in u for all the processes.*/
      MPI_Reduce(&dumax_local,&dumax,1,MPI_DOUBLE,
                         MPI_MAX,0,MPI_COMM_WORLD);
/*Print out the maximum change in u for all the processes.*/
      if (myrank==0) printf("it = %2d  dumax = %12.8f \n", it
    } /* end of it loop */
   MPI_Finalize();
   return 0;
}
```

Compute local u(x^n) and maximum local change.

Update local u(x).

Gather the maximum change and print at root. Continue it loop

ERDC MSRC Major Shared Resource Center

How can threads be used?

- What issues need to be considered when using OpenMP with MPI to create a multilevel parallel Laplace equation solver?
  - *Are there any computationally intensive areas of the code?* The loop that computes the local value of the array u could contain a large number of iterations.
  - *Are there any data dependencies within the computationally intense areas?* The loop over the convergence iterations cannot be parallelized with threads. Each iteration has to be complete before the next one begins. However, the loop over the local arrays can be parallelized with threads.
  - *What would be the easiest way to create a dual-level parallel code?* In this case with a single loop, adding OpenMP parallel constructs is very easy.
  - *How do you check for correctness and efficiency?* Compare the final answers to the serial answers and use some timers to compute the codes efficiency.

ERDC MSRC Major Shared Resource Center

Adding OpenMP Directives: First Pass

```
for (it=0; it<itmax; it++)
{
     thread_id = omp_get_thread_num();
     dumax[thread_id] = 0;

#pragma omp parallel for shared(du,dumax)
     private(i,istart,istop,d)
     for (i=istart; i<=istop; i++) {
        d     = 0.5*(u[i+1]-2.0*u[i]+u[i-1]);
        du[i] = d;

        d = d<0 ? -d:d;
        if (dumax[thread_id] < d)
          dumax[thread_id] = d;
       }

#pragma omp parallel for shared(u,du)
     private(i,istart,istop)
     for (i=istart; i<stop; i++) {
        u[i] += du[i];
       }

} /* end of the it loop */
```

First parallel region.

Second parallel region.

ERDC MSRC Major Shared Resource Center

Adding OpenMP Directives: Better

```
#pragma omp parallel shared(u,du,itmax,imax,dumax)
     private(it,i,istart,istop,thread_id,d)
  {
   for (it=0; it<itmax; it++) {
     thread_id = omp_get_thread_num
     dumax[thread_id] = 0;

#pragma omp for
     for (i=istart; i<=istop; i++) {
        d     = 0.5*(u[i+1]-2.0*u[i]+u[i-
        du[i] = d;

        d = d<0 ? -d:d;
        if (dumax[thread_id] < d)
          dumax[thread_id] = d;
       }


#pragma omp for
     for (i=1; i<imax-1; i++) {
        u[i] += du[i];
       }

     } /* end of the it loop */
  } /* end of the parallel region */
```

Go ahead and put a parallel directive around the it loop:

- All threads will execute the it loops.
- Threads will not be created and destroyed many times over.
- Same result, with less overhead.

Adding OpenMP Directives: What Else?

```
/* Initialize the arrays.*/
#pragma omp parallel
      shared(u,du,itmax,imax,duma
       private(i, imax_local)
  for (i=0; i<imax_local; i++) {
      u[i]  = 0.0;
      du[i] = 0.0;
    }
```

Initialize the arrays within an OpenMP directive.

On those machines where the location of memory is an issue, this could make a significant difference.

Data Locality.

ERDC MSRC Major Shared Resource Center

What to look for when the speedup is not what you expected?

- Are threads being created and destroyed over and over?
  - Move the creation of the threads farther out in the algorithm.
  - Use environment variables to keep the threads in a spin state.
- Are the data actually located far away from the processor on which a thread is running?
  - Initialize the arrays within a threaded region.
- Is there enough work for threads to be useful?
  - Typically, on the order of 1000s of floating-point operations are needed within a region to make the creation of threads worthwhile.
  - Can you use fewer numbers of MPI processes to make the threads more efficient? You might actually get a better time.

ERDC MSRC Major Shared Resource Center